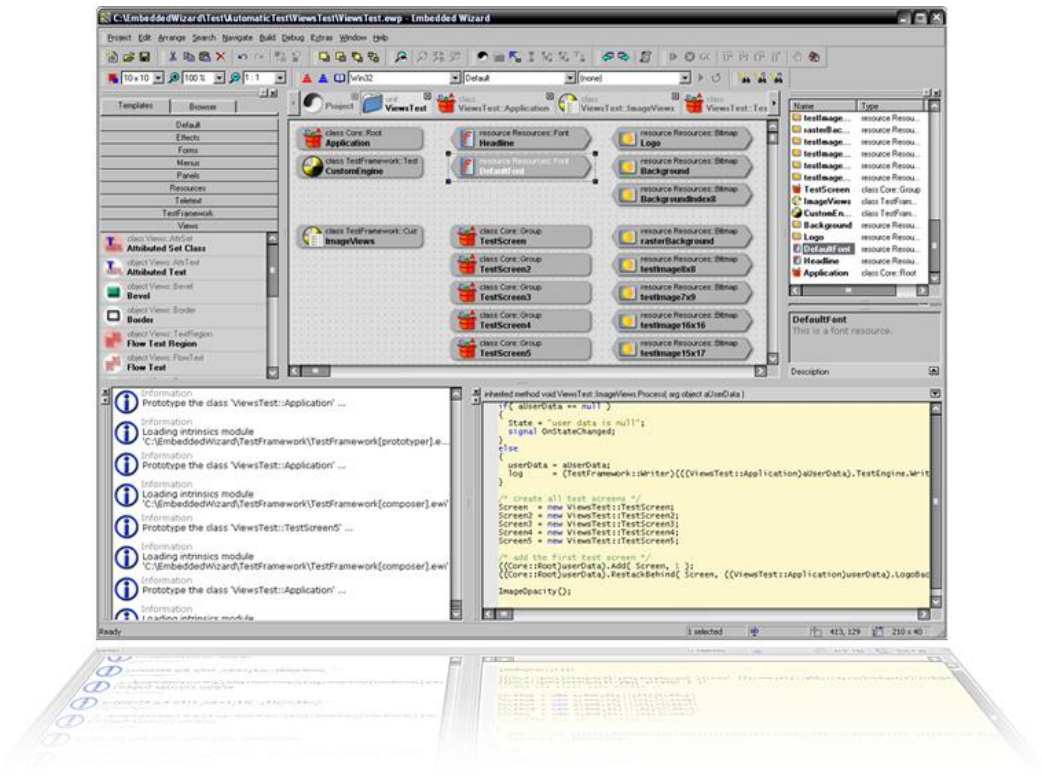
Embedded Wizard

# Chora User Manual



Version 6.51
28 April 2015

Authors: Dipl. Ing. Paul Banach, Dipl. Ing. Manfred Schweyer

Copyright (c) TARA Systems GmbH

# Contents

1 Introduction ..... 4

2 Programs in Chora ..... 7

3 Project file (EWP) ..... 9

3.1 Units ..... 9
3.2 Profiles and macros ..... 10
3.3 Languages ..... 15
3.4 Styles ..... 16
3.5 Comments ..... 17
3.6 Example ..... 18

4 Unit file (EWU) ..... 19

4.1 Classes ..... 20
4.1.1 Instance variables ..... 23
4.1.2 Instance arrays ..... 26
4.1.3 Properties ..... 29
4.1.4 Embedded objects ..... 34
4.1.5 Methods ..... 36
4.2 Constants ..... 54
4.3 Resources ..... 56
4.3.1 Attributes of a bitmap resource ..... 59
4.3.2 Attributes of a font resource ..... 61
4.4 Auto objects ..... 63
4.5 Inline code ..... 65
4.6 Enumerations ..... 67
4.7 Sets ..... 69
4.8 Variants ..... 71
4.8.1 Class Variants ..... 72
4.8.2 Constant Variants ..... 74
4.8.3 Resource Variants ..... 76
4.8.4 Auto object Variants ..... 79
4.9 Comments ..... 81
4.10 Example ..... 82

5 Statements ..... 84

5.1 '`var`' declaration statement ..... 85
5.2 '`array`' declaration statement ..... 87
5.3 Expression statement ..... 88
5.4 Empty statement ..... 89
5.5 Compound statement ..... 89
5.6 '`if`' selection statement ..... 90
5.7 '`switch`' selection statement ..... 91
5.8 '`for`' iteration statement ..... 94
5.9 '`while`' iteration statement ..... 95


Copyright TARA Systems GmbH
www.embedded-wizard.de

5.10 'return' statement ... 96
5.11 'signal' statement ... 97
5.12 'postsignal' statement ... 98
5.13 'idlesignal' statement ... 99
5.14 'throw' exception statement ... 101
5.15 'trace' debug statement ... 102
5.16 'tracestack' debug statement ... 102
5.17 'native' statement ... 102
5.18 'attachobserver' statement ... 105
5.19 'detachobserver' statement ... 108
5.20 'notifyobservers' statement ... 109

6 Data types ... 112

6.1 Instant data types ... 112
6.2 User defined data types ... 117
6.3 Reference data types ... 118

7 Operators ... 120

7.1 Unary instant operators ... 121
7.2 Binary instant operators ... 123
7.3 Instant cast operators ... 133
7.4 Object cast operators ... 135
7.5 'enum' and 'set' cast operators ... 138
7.6 'new' operator ... 139
7.7 Instant constructors ... 141
7.8 Instant methods ... 144
7.9 Instant properties ... 147
7.10 Index operator ... 156
7.11 Assignment operator ... 158
7.12 'parentthis' operator ... 159
7.13 'classof' operator ... 159
7.14 Build-in functions ... 159

8 Chora preprocessor ... 161

8.1 Macro evaluation ... 161
8.2 Preprocessor directives ... 167
8.3 Control directives ... 170

9 Language selection ... 174

10 Usage of Variants ... 179

11 Garbage collection ... 185

12 Comments ... 187

13 Optimization ... 190


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 1 Introduction

Chora is a universal object-oriented programming language for developing graphical user interfaces (GUIs), and so-called on-screen displays (OSDs) for embedded systems. Programmers use Chora to describe the appearance and behavior of the GUI. It defines the position, size, color and font of graphical GUI objects and describes how these objects should communicate with one another and react to user interaction. A special Chora compiler then translates the GUI description into the ANSI C language and enables simple integration of the GUI within an embedded system.

The Chora programming language is not restricted to any specific target platform. An GUI created with Chora is therefore platform-independent and can be ported to another system with a minimum of effort. A limiting factor is the graphical performance of the embedded systems used. Depending on the color depth and screen resolution supported by the target system, it may be necessary to adapt the appearance of the GUI when porting to a new target system.

The structure and syntax of Chora are largely based on the programming languages C, C++ and Java. This closeness simplifies and shortens the time needed to familiarize oneself with Chora. Still, Chora is not C++ and it is not Java. Many language constructs, such as the pointers and pointer arithmetic popular with C and C++ programmers are deliberately not supported in Chora. These restrictions serve foremost to limit possible sources of error and to enable development of a platform-independent GUI. Further, these restrictions make it easier for non-C, C++ and Java programmers to become familiar with Chora.

As an object-oriented programming language, Chora supports the following important object-oriented paradigms:

- Attributes → variables of an object in which the state of the object can be stored (e.g. text color). Attributes are often called 'fields'. → see "Instance variables" (chapter 4.1.1) and "Instance arrays" (chapter 4.1.2).
- Methods → functions of an object. By calling a method, the object can be accessed. For example, an object can be drawn on screen by calling its `Draw()` method. → see "Methods" (chapter 4.1.5).
- Properties → are intelligent attributes of an object. Unlike normal attributes, each property has an onget and an onset method. Accessing the property invokes the appropriate onget or onset method in a way that is completely transparent to the programmer. → see "Properties" (chapter 4.1.3).
- Encapsulation → The value of a property can be accessed only via its onget or onset methods. In this way, the contents of the property are protected from access outside the object.
- Class derivation → The class of an object can be derived from another class, taking on the attributes and methods of the base class. In the derived class, the attributes or methods taken on may be overridden and adapted. → see "Classes" (chapter 4.1).


Copyright TARA Systems GmbH

- Polymorphism → All methods of a class are virtual. This means that the method of an object suited for the invocation is determined only at the runtime. From the perspective of the calling method, an object can thus take varying behavior — the object is polymorphic. For example, a `Draw()` method used in scope of a text object draws its text on the screen. By contrast, the `Draw()` method of a rectangle object draws a rectangle.

The use of the object-oriented paradigm increases the reusability, maintainability and platform-independence of the GUI software developed and also reduces the size of the code.

Besides the object-oriented paradigm, the Chora programming language is distinguished by many other characteristics necessary in the development of GUI applications:

- Objects no longer in use are destroyed automatically and memory used by them is automatically released, thanks to the integrated Garbage Collector. → please refer to the chapter titled "Garbage collection" (chapter 11).
- Support for resources such as bitmaps or fonts that are required in an GUI application. → please refer to the chapter titled "Resources" (chapter 4.3).
- Support for the development of multilingual GUI applications. A multilingual GUI application may contain text, images, etc. for different languages. → please refer to the chapter titled "Language selection" (chapter 9).
- Support for class, constant and resource variants. The variants allow the development of GUI applications, whose appearance and behavior can be changed at the runtime. This unique feature simplifies the implementation of skins, themes, target and resolution dependent GUIs, etc. This feature also provides the customization of existing software in a way very transparent to the customer → please refer to the chapter titled "Variants" (chapter 4.8) and "Usage of Variants" (chapter 10).
- A comprehensive set of supported data types, from simple integer data types (`int8`, `int16`, ...) to complex data types such as `string`, `color`, `rect`, etc. → please refer to the chapter titled "Instant data types" (chapter 6.1).
- A comprehensive set of operators that can be used to evaluate expressions, from simple arithmetic/logical operators ( '+', '–', etc.) to complex instant methods, instant constructors, object cast, etc. → please refer to the chapter titled "Operators" (chapter 7).
- User-defined enumeration and set data types. → please refer to the chapters titled "Enumerations" (chapter 4.6), "Sets" (chapter 4.7) and "User defined data types" (chapter 6.2).
- Instructions to control the program flow, `for` and `while` loops, local variables, etc. → see "Statements" (chapter 5).
- Signals and slots for simple communication between objects. An object may send signals to slots of other objects without necessarily knowing the identity of the recipient object. → see "'signal' statement" (chapter 5.11) and "slot methods" (chapter 4.1.5.4).


Copyright TARA Systems GmbH
www.embedded-wizard.de

- Pending signals for deferred code execution. The signals are delivered before or after the screen update is performed – depending on the used statement → see "'postsignal' statement" (chapter 5.12) and "'idlesignal' statement" (chapter 5.13).
- Reference data types are also useful for communication between objects. With the aid of a reference, it is possible to access a property of an object without necessarily knowing the identity of that object. → see "Reference data types" (chapter 6.3).
- Simple and powerful infrastructure for registering of observers and delivering of notifications to them when the observed subject has been signaled. This technique provides the basis for the Controller/View model and is used in the development of complex GUI applications → see "'attachobserver' statement" (chapter 5.18), "'detachobserver' statement" (chapter 5.19) and "'notifyobservers' statement" (chapter 5.20).
- Auto objects, which are instantiated automatically on-request and released, when they are not in use anymore. Unlike the ordinary Chora objects, auto objects do exist in the global scope, so they can be directly accessed from anywhere of the Chora code. Usually auto objects do serve as 'controller' in the Controller/View model and is used in the development of complex GUI applications → see "Auto objects" (chapter 4.4).
- A Chora preprocessor enables conditional code generation, depending on the target system or profile chosen. → see "Chora preprocessor" (chapter 8) and "Profiles and macros" (chapter 3.2).
- Support for native methods and inline code. In this way, the platform-independent Chora program code may be combined with the platform-dependent native code of the target system. → see "Native methods" (chapter 4.1.5.1), "'native' statement" (chapter 5.17) and "Inline code" (chapter 4.5).

The Chora programming language with its GUI model forms the basis for the Embedded Wizard.


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 2 Programs in Chora

A program in Chora can be developed using an ordinary text editor, preferably one capable of syntax highlighting. This simplifies the development of a program in Chora.

A Chora program consists of at least two files:

- Project file → Contains the description of the entire project. Here the settings for the desired target platform are specified. A project file controls the Chora compiler and can be compared to a make file. A project file has the extension `.EWP`.
- Unit file → Contains the actual definition of the GUI. In the unit file, the classes, methods, attributes and so forth are specified. A unit file can be compared to a C or C++ file or to a Java package. A unit file has the extension `.EWU`.

A program in Chora may contain multiple unit files. The unit files used must be listed in the project file, so that the Chora compiler may use them during the translation. The possibility of splitting a Chora program into multiple unit files also increases the reusability and maintainability of the software and further makes it possible for multiple programmers to work simultaneously on the same GUI.

To translate a Chora program to ANSI C, the Chora compiler `CHORAC.EXE` must be launched with the name of the desired project file. For example:

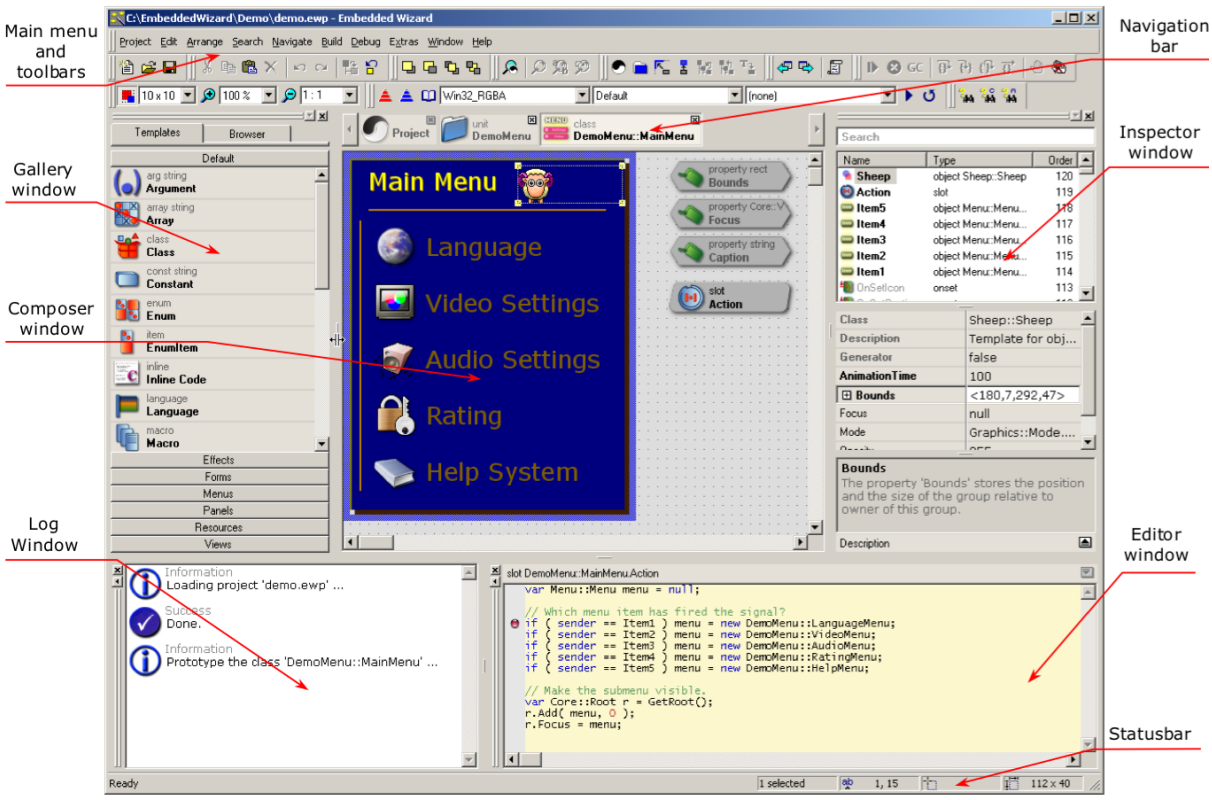
```
chorac   Elixir.ewp   Win32
```

The second argument, `Win32`, defines the name of the desired profile. A profile contains the definition of the target system. In a project file, multiple profiles may be contained, each for a different target system. When the Chora compiler is launched, a profile must be selected explicitly in order to set the Chora compiler to the correct target platform.

If this is successful, the Chora compiler creates a directory with the same name as the profile chosen and stores the automatically generated ANSI C files in this directory. The names of the generated files are derived from the names of the unit files, classes, definitions, etc.

If the Chora program can't successfully be translated, all errors and warnings are reported, so that the programmer may correct the cause of the error in the Chora program.

The more comfortable way to develop Chora programs is provided by the Embedded Wizard IDE (Integrated Development Environment). Embedded Wizard IDE provides a very effective environment for the development of GUI applications. It is based on the idea of visual programming, where software components, so called 'bricks' are arranged and connected with the mouse by simple 'drag & drop'. From our point of view it was self-evident to combine the visual programming and the GUI application development – the both concepts complement one another. The following figure shows the Embedded Wizard Main Window at work:




The software development with the Embedded Wizard is completely object oriented and based on the idea of components. The components are encapsulated, independent building blocks, which can be combined with other components to a complex GUI. The modeling of components is done with Embedded Wizard IDE in a visual way. To implement the functionality of components, a large set of Chora statements is available.

The very big advantage of the Embedded Wizard IDE is the integrated prototyping environment. It allows the testing and presenting of the complete GUI application already during the development on the PC, without the necessity to download of the application into the target system!

This document contains the detailed specification of the Chora programming language. Usually, when you are working with the Embedded Wizard IDE, you don't need to know all details of this specification. The Embedded Wizard IDE hides the complexity of the GUI application development and makes it much more easy. Instead of writing a lot of Chora code, you can concentrate on the graphical aspects of your GUI application. For more details about the Embedded Wizard IDE see the document "Embedded Wizard User Manual".


Copyright TARA Systems GmbH

# 3 Project file (EWP)

Each Chora program must be described by a project file. This file specifies what a Chora program consists of and how the Chora compiler should ultimately translate it to ANSI C.

A project file must always begin with a `$version` directive. This directive informs the Chora compiler of the syntax used in the project file, and is intended for the future file extensions. The directive is followed by the version number. For example:

```
$version 5.0
```

If the Chora compiler does not recognize the version number given, the translation is aborted with an error message.

After the `$version` directive follows the actual listing of the project's components. These include:

- Units → Each unit file used in the Chora program must be defined explicitly in the project file. Only in this way can the Chora compiler access all relevant units. The definition of a unit comprises the indication of where within the file system the `.EWU` unit file may be found.
- Profiles → A profile describes the settings of the desired target platform. The target platform is determined from the name of the so-called Platform Packages. A Platform Package is an Embedded Wizard extension that must be supplied by TARA Systems GmbH for each target platform. Besides the name of the Platform Package used, a profile may contain a set of settings and macro definitions.
- Macros → A macro stores any character string under a unique reference name. By using this reference name, the contents of the macro (the character string stored there) can be used in definitions of classes, methods, and so on. A macro is always defined within a profile; thus, a macro is valid only when the corresponding profile has been specified when launching the Chora compiler. Macros defined in other profiles are ignored.
- Languages → Defining languages determines the countries for which the GUI is localized. The localization makes it possible to develop an GUI in German, English, Japanese and so on.
- Styles → The styles are used to identify different variants of the GUI application. Such multivariant GUI application may change its appearance and behaviour at the runtime. This feature provides a simple way for the development of GUI applications with different skins and themes.

## 3.1 Units

A unit is always introduced using the keyword `unit`, followed by the name of the unit itself:

```
unit Core
{
  attr Directory = ..\Mosaic;
}
```


Copyright TARA Systems GmbH

This defines a unit named Core. The Chora compiler will attempt to read the unit file `Core.ewu`, in order to generate the ANSI C code from it. In so doing, the compiler uses the attribute `Directory` in order to find the directory that contains the `Core.ewu` file. In the example above, the unit file is searched for in the directory `..\Mosaic`, relative to the project directory, where the project file itself is stored.

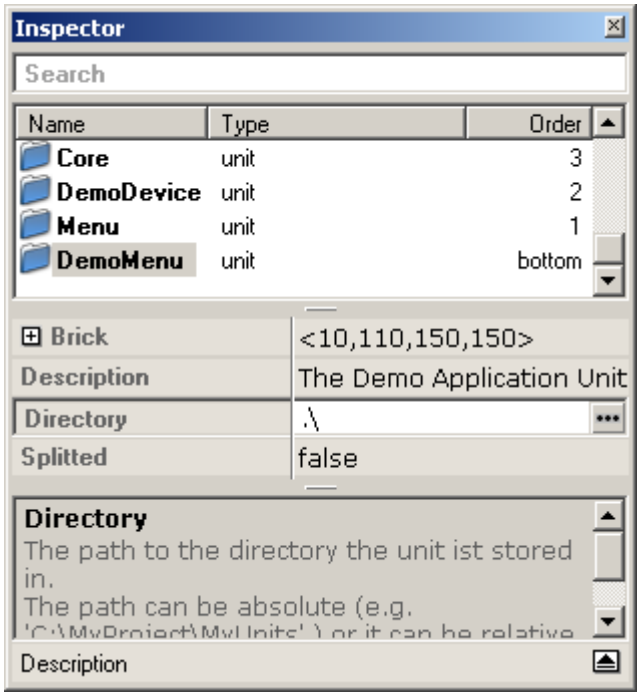
Note the semicolon '`;`' behind the attribute `Directory`. The semicolon closes the attribute. Behind the semicolon, there may be no further characters before the line break.

The unit must have a unique name. There may not be multiple units with the same name in a project file. It is also not possible to use the same name for a unit as well as for a profile, a language or a style.

Please note that each project must define at least one unit.

Within the Embedded Wizard IDE the units appear as bricks in the Project Composer window:

A double click on the unit brick will show its content. The attributes of the unit can be set directly in the Inspector window, when the appropriate unit brick has been previously selected:




## 3.2 Profiles and macros

A profile is always introduced using the keyword `profile`, followed by the name of the profile itself:

```
profile Win32 : Tara.Win32.RGBA8888;
```

Here, a profile named `Win32` is being defined. This profile uses the Platform Package `Tara.Win32.RGBA8888` (MS Windows in graphics mode RGBA 8:8:8:8 (32 bits per pixel)). The Chora compiler will then have to generate code for the `Tara.Win32.RGBA8888` platform, assuming that the profile `Win32` is specified when launching the compiler:

```
chorac   Elixir.ewp   Win32
```

The primary function of the profile is to cover the code generation settings for the particular target system. The Code Generator evaluates these settings in order to decide how to generate the code for this target system. The settings are introduced by the `attr` keyword and listed within the braces block of the profile:

```
profile Win32 : Tara.Win32.RGBA8888
{
  attr Optimization     = High;
  attr Obfuscation      = false;
  attr ScreenSize       = <640,480>;
  attr Verbose          = false;
  attr Clut             = defaultclut.txt;
  attr OutputDirectory  = c:\temp\emwisrc;
  attr PostProcess      = nmake /f makefile.mak;
  attr ApplicationClass = Example::Application;
  attr ApplicationTitle = "Hello World Example";
  attr ApplicationName  = HelloWorld;
}
```

| Attribute name | Description |
|---|---|
| `Clut` | The name of the Color Lookup Table (CLUT) file containing the definition of colors used within this GUI application. The CLUT is used for `Index8` platforms only. In all other cases, the value of this attribute is simply ignored. If no CLUT file is specified, the default file `DefaultClut.txt` is used.<br><br>To access the value of this attribute at the runtime, the build-in macro `$Clut` is available. |
| `Optimization` | The optimization level attribute controls the code generation. Depending on the value of this attribute, the Code Generator may perform different code simplification and elimination steps. Following levels are available:<br><br>➢ `None` No optimization is performed.<br><br>➢ `Low` Performs only simple optimization steps. All invocations of non-overridden methods are performed directly, without the VMT indirection. The read/write access to properties is performed without the onget/onset method invocations.<br><br>➢ `Medium` Forces the Code Generator to reduce the memory usage of Chora objects; the layout of Chora objects is optimized.<br><br>➢ `High` The highest level of optimization forces the Code |


Copyright TARA Systems GmbH

| | |
|---|---|
| | Generator to eliminate all unreferenced class members (variables, methods, etc.).<br><br>The automatic elimination of a class member can be explicitly suppressed by using the `$output true` directive in front of the class member's definition.<br><br>To access the value of the `Optimization` attribute at the runtime, the build-in macro `$optimization` is available.<br><br>For more details see "Optimization" (chapter 13). |
| `Obfuscation` | The obfuscation attribute controls the format of the generated code. If this attribute is `true`, the Code Generator will try to compress and obfuscate the generated code. If this attribute is set `false`, the code is not modified.<br><br>The effect of this attribute depends on the used Code Generator. If the generator doesn't support the code obfuscation, then the attribute has no effect on the generated code.<br><br>This attribute can take the values `true` or `false` only. The value of this attribute may be evaluated in the Chora code using the build-in macro `$obfuscation`. |
| `ScreenSize` | The size of the screen in pixel provided for this profile. Embedded Wizard expects here a valid Chora `point` literal, e.g. `<720,576>`. The screen size determines the visible area of the entire GUI application.<br><br>To access the value of this attribute at the runtime, the build-in macro `$ScreenSize` is available. |
| `ApplicationClass` | The name of the user defined application class. The application class serves as the root of the GUI application. Usually application classes are derived from the Mosaic class `Core::Root`.<br><br>To access the value of this attribute at the runtime, the build-in macro `$ApplicationClass` is available. |
| `ApplicationTitle` | The title of the GUI application as a string literal e.g. `"Solar Power Plant Control Panel"`.<br><br>The application title describes the GUI application. The usage of this title depends on the target system. For example, in the `Tara.Win32` target the title is shown in the window caption.<br><br>To access the value of this attribute at the runtime, the build-in macro `$ApplicationTitle` is available. |
| `ApplicationName` | User defined, optional application name as an identifier (e.g. `DashboardApp`). This attribute allows you to distinguish the application within an environment executing simultaneously multiple Embedded Wizard applications.<br><br>The effect of this attribute depends on the platform package |

| | |
|---|---|
| | used for the code generation. If the generator doesn't support the code obfuscation, then the attribute has no effect on the generated code.<br><br>The value of this attribute can be accessed from Chora by using the build-in macro `$ApplicationName`." |
| `Verbose` | The verbose attribute controls the amount of comments included into the generated code. If this attribute is `true`, the Code Generator generates more detailed comments. If this attribute is set `false`, only the most important comments are included. The effect of this attribute depends on the used Code Generator.<br><br>To access the value of this attribute at the runtime, the build-in macro `$verbose` is available. |
| `OutputDirectory` | The path to the destination directory where the generated files should be stored in. The path can be absolute (e.g. `C:\temp\emwisrc`) or it can reside relative to the directory of the project file (e.g. `..\output`).<br><br>This attribute is optional. If no value is specified, the Embedded Wizard per-default stores the generated files into a sub directory with the name of the appropriate profile. |
| `OutputPrefix` | This attribute specifies an optional prefix for the generated file names. The value of this attribute is only a suggestion for the Code Generator how the generated files should be named. If the Platform Package is not able to evaluate this attribute, the prefix will be ignored.<br><br>This attribute was intended in order to support the customer specific build processes. Please note the usage of this attribute influences the interface to the generated code whereby compilation errors might occur. |
| `PostProcess` | The user defined command, which will be executed at the end of the build process. The command can be an `EXE`, `BAT` or a `CMD` file and it can be followed by any arguments (e.g. the command to start the make job after the code generation is finished `nmake.exe /f makefile.mak`).<br><br>This attribute is optional. If specified, Embedded Wizard searches for the command file, starts it in a new process and waits for its termination. Any outputs of the command will be stored in a log file in the destination directory of the current profile. The working directory of the command is per-default set to the current project directory.<br><br>If the command is specified without a path, or the specified path is relative, Embedded Wizard first searches for the command file in the current project directory and then in the Embedded Wizard installation directory. In case of an `EXE` file the Windows directory and the directories specified in the `PATH` environment variable are evaluated in order to find the |

| | |
|---|---|
| | affected application.<br>Please note the attribute expects, that the command has the appropriate file extension specified. |

**Tabelle 3-1**

Beside the code generation settings, the profile may contain one or more macro definitions. These definitions must be listed in braces and made subordinate to the profile in the same manner as it is done with the attributes:

```
profile Win32 : Tara.Win32.RGBA8888
{
  macro Debug        = true;
  macro DefaultFont  = Verdana;
}
```

In this case, the profile was extended by the two macros `Debug` and `DefaultFont`. Both of these macros are available only if the Chora compiler was launched with the profile `Win32`. In other profiles, the macros are either undefined or they contain other values. The profile-dependent definition of macros allows the programmer to include profile settings in the definition of classes, methods, etc.

Of course the attributes and the macros may be combined together within the profile's definition:

```
profile Win32 : Tara.Win32.RGBA8888
{
  attr Optimization     = High;
  attr ApplicationClass = Example::Application;
  attr ScreenSize       = <640,480>;
  attr Verbose          = false;
  attr Clut             = defaultclut.txt;
  macro Debug           = true;
  macro DefaultFont     = Verdana;
}
```

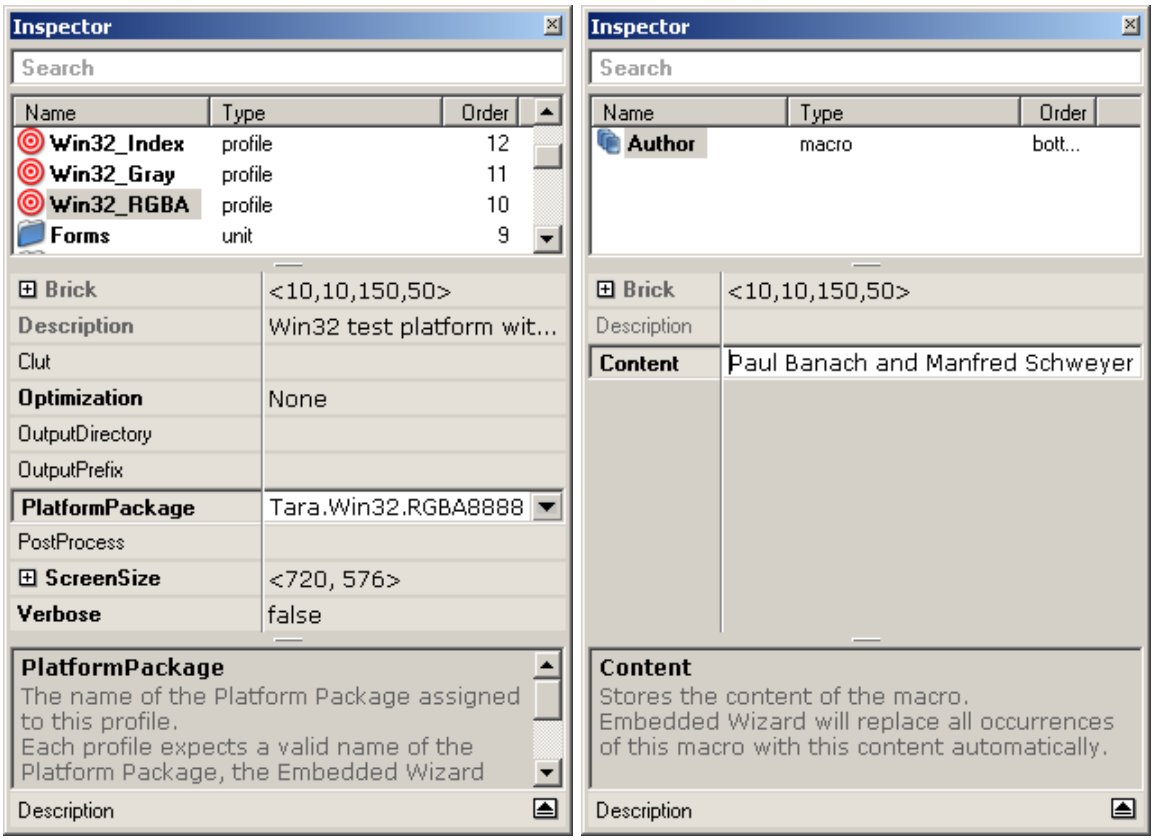
Please note that each project must define at least one profile.

The profile name must be unique. Multiple identically named profiles may not be defined in a project file. It is also not possible to use the same name for a unit and for a profile, a language or a style.

The name of a macro must, however, be unique only within a particular profile. This means that a macro may have the same name as a profile or a unit. It is not allowed for two identically named macros to be defined in the same profile.

Within the Embedded Wizard IDE profiles and macros appear as bricks in the Composer window:

A double click on the profile brick will show the macros stored within it. The attributes of the profile or the content of the macro can be set directly in the Inspector window, when the appropriate profile or macro brick has been previously selected:




## 3.3 Languages

A language is always introduced by the keyword `language`, followed by the name of the language itself:

```
language English;
```

This defines a language named `English`. The name of the language can now be used in the definition of constants and resources, in order to label the language dependency.

Please note that each project must define at least one language that has the name `Default`, in which `Default` fulfills a special purpose. The default language will be used for all non-language-dependent GUIs.

More details can be found in "Language selection" (chapter 9).

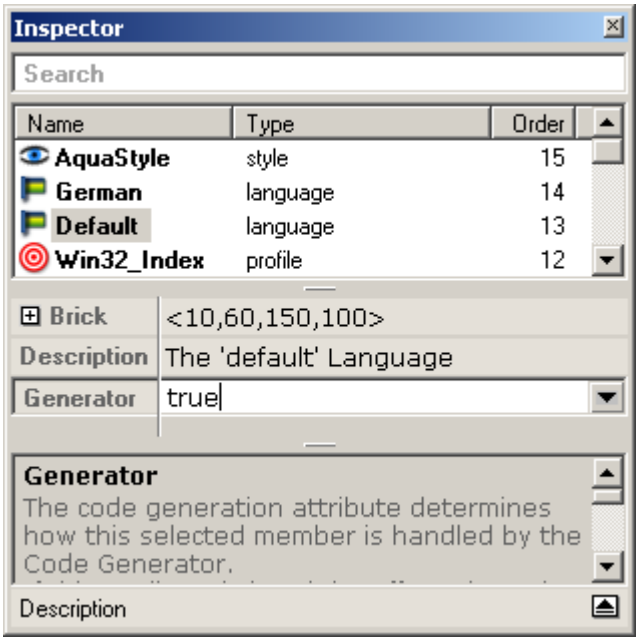
Optionally, the `language` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
language English;
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE languages appear as bricks in the Project Composer window:

The attributes of the language can be set directly in the Inspector window, when the appropriate language brick has been previously selected:




## 3.4 Styles

A style is always introduced by the keyword `style`, followed by the name of the style itself:

```
style Aqua;
```

This defines a style named `Aqua`. The name of the style can now be used in the definition of a class, constant or resource variant. For this purpose the variants accept a variant condition directive, which determines, when the origin class, constant or resource is substituted by the affected variant. In this manner, an GUI application may contain multiple, style dependent appearance and behaviour. More details can be found in "Variants" (chapter 4.8) and "Usage of Variants" (chapter 10).

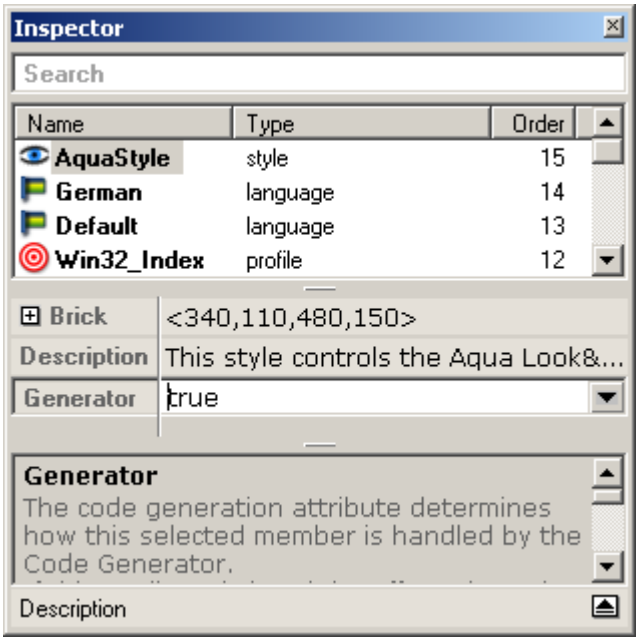
Optionally, the `style` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
style Aqua;
```

The `$output` directive of a style does also affect the dependent variants. If such a variant does depend on disabled styles only, the variant is ignored by the code generation. For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE styles appear as bricks in the Project Composer window:

The attributes of the styles can be set directly in the Inspector window, when the appropriate style brick has been previously selected:




## 3.5 Comments

Comments may be added to all units, profiles, macros, languages and styles. Such comments are preceded by a double slash `//` and extend to the end of the line of code:

```
// The unit 'Core' contains base classes: View, Group,
// Root and Timer.
unit Core
{
  attr Directory = ..\Mosaic;
}
```

Unlike other programming languages, the comments given are not ignored or simply bypassed. Instead, they are read by the Chora compiler and output in the generation of the ANSI C code. From Chora's standpoint, comments are fully valid syntactic elements. For this reason, it is not allowed to write a comment just anywhere in a project file. This excerpt from an erroneous project file demonstrates this:

```
unit Core
{
  attr Directory = ..\Mosaic; // Misplaced comment!!!
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

A comment must always directly precede the definition to which it applies. Comments are very useful to document the functionality of units, languages, profiles, etc.. Embedded Wizard uses the content of comments to give you a short description of members, which have been selected in the Composer window of the Embedded Wizard application. This description appears below the Inspector window. For more details see 'Embedded Wizard User Manual'.

Additionally Embedded Wizard is able to generate a documentation file with the description of your project including the class hierarchy, declaration of classes, methods, properties, constants, resources, etc. This automatic generated documentation file can then be used or distributed as 'software reference manual' of your project. Further details to comments and their syntax can be found in "Comments" (chapter 12).

## 3.6 Example

The following example shall demonstrate the structure of a project file. The project consists of 2 units, of 1 profile and 2 languages:

```
$version 5.0

// The unit 'Core' contains base classes: View, Group,
// Root and Timer.
unit Core
{
  attr Directory = ..\Mosaic;
}

// This is the unit containing menus, menu items, etc.
unit Menu
{
  attr Directory = .\;
}

// The profile Win32 defines a profile for the MS Windows
// target with the given resolution of 800x600 pixels.
profile Win32 : Tara.Win32.RGBA8888
{
  attr ApplicationClass = Menu::Application;
  attr ScreenSize = <800,600>;
  attr Optimization = High;

  // Defines, whether the EPG is enabled or not.
  macro EnableEPG = true;
}

language German;
language Default;
```

# 4 Unit file (EWU)

Each Chora program contains at least one unit file. In the unit files, the classes, their methods, and so on, are defined. Which unit files belong to a project is specified in the project file.

A unit file must always begin with `$version` directive. This directive informs the Chora compiler of the syntax used in the unit file and is intended for the future file extensions. The directive is followed by a version number. For example:

```
$version 5.0
```

If the Chora compiler does not recognize the version number given, the translation is aborted with an error message.

Behind the `$version` directive, there follow the actual definitions. These include:

- Classes → The definition of a class contains the description of all attributes and all methods that belong to a class. Thanks to the class derivation, a class has to define only the new or changed attributes, or only the methods overridden. No other attributes or methods that are taken over unchanged in the derived class need to be listed.
- Constants → A constant bears a name and has a fixed value defined by the programmer. Using this name, the value of a constant can be used in the Chora program. Constants are excellently suited to defining, for example, menu titles or the colors of an GUI. Changing a constant, therefore, affects the entire GUI.
- Resources → Resources are constant files that the Chora compiler adopts in the ANSI C code generated. Thus it is possible, for example, to embed bitmaps or fonts in the GUI in order to use them at runtime from the GUI and display them on the screen.
- Inline Code → Inline is a special definition that makes the Chora compiler insert a native code sequence in the generated ANSI C code. In this manner, the combination of Chora and ANSI C code is possible. This technique is used primarily in developing driver classes geared to specific platforms. Such driver classes form the interface between the GUI and the underlying target system, and guarantee the platform independence of the GUI that has been developed.
- Enumerations → An enumeration is a user-defined data type that consists of a list of constant labels, so-called enumerators. A variable of the type of such an enumerator may, therefore, take on exactly one value from the enumerator list. The Chora compiler is very strict in handling enumeration data types and prevents the programmer from assigning to a variable of a particular enumeration type a value that has not been explicitly defined in the enumeration. Except for the strictness of the Chora compiler, an enumeration can be compared to a C, C++ `enum`.
- Sets → A set functions in a similar way to an enumeration. The only difference to an enumeration is that the elements of a set may be combined in any way. This means that a variable of type 'set' may contain any combination of elements of the set, including an empty set (containing no elements).


Copyright TARA Systems GmbH
www.embedded-wizard.de

- Auto objects → Objects, which are automatically instantiated on-request and released as soon as they are not in use anymore. Auto objects always exist in the global scope of a unit so they can be accessed from any part of the Chora code.

Optionally, the unit file can be split in several include files, one file for each constant, resource, class, etc. definition. Include files, which belong to a unit have to be stored within a directory with the name of this unit. While loading a project, Embedded Wizard determines, whether a unit is stored as a single file or as a directory consisting of several include files.

The name of an include file has to match the name of the corresponding definition followed by the file extension `*.EWUI`. For example, the definition for the class `Core::Timer` will be stored in the include file `Timer.ewui` within the directory `Core.ewu`.

Similar to other Embedded Wizard files, include files should start with the `$version` directive followed by the appropriate definition:

```
$version 5.0

// Description of the class.
class Timer
{
  ...
}
```

An additional `.DIR` file within the directory has to contain the list of the appropriate include files. This file should also start with the `$version` directive followed by a list of `$include` directives, one for each include file (Note the file names are without the extension):

```
$version 5.0

$include View
$include Group
$include Root
$include Timer
```

The unit splitting improves the simultaneous development of large units within a team of developers. In this case, each developer can work with it's own class, constant, resource, etc. include file. Only the `.DIR` file may need a manual adaptation in order to consolidate changes made by different developers.

## 4.1 Classes

Classes are always introduced by the keyword `class`, followed by the name of the class:

```
class Shape
{
  var rect  Bounds = <0,0,128,64>;
  var color Color  = #FF0000FF;

  method void Draw()
  {
    // Do something to draw the shape ...
  }
}
```

Using this definition, we have created a simple class containing two variables (attributes): `Bounds` and `Color`, as well as one method, `Draw()`. This class was derived from no other base class. That means that aside from these two variables and the single method, no other members are contained in this class.

Once it is defined, the class serves as a template for creating objects. The programmer can, at runtime, create as many objects of a class as he likes, call methods of the objects or directly access the objects' attributes:

```
var Sample::Shape theObject = new Sample::Shape;
theObject.Draw();
```

The above example demonstrates how a new object of class `Sample::Shape` is created. Note that the complete name of a class requires the name of the unit in which the class was defined. Thus, the `Shape` class was defined in the `Sample` unit. If the unit name is missing, the Chora compiler will not be able to find the class.

After its creation, the object can be accessed. In the above example, the `Draw()` method of this object is called, making the object, for example, appear on the screen. What exactly happens when the method is called depends, of course, on how it is implemented.

A special quality of classes is the ability of one class to 'inherit' its definition from another class. In so doing, the definition of the base class is taken over completely by the new class. The inherited definition can then later be extended or changed without influencing the base class. To derive from a class, the name of the base class must be given explicitly in the definition:

```
class Rectangle : Sample::Shape
{
  inherited method Draw()
  {
    // Here we can draw the rectangle ...
  }
}
```

Each inherited class thus takes on the complete definition of the base class. In this case, for example, the `Rectangle` class inherits the `Bounds` and `Color` variables as well as the `Draw()` method from the base class `Sample::Shape`. The `Draw()` method was even overridden in order to allow the `Rectangle` class to do something that deviates from the generic `Shape` class. If, then, at runtime the `Draw()` method is called in the context of a `Rectangle` object, the program automatically branches into the appropriate version of the method.


Copyright TARA Systems GmbH
www.embedded-wizard.de

The depth of the class inheritance is not limited by the Chora compiler. Thus it is possible, when using Chora, to develop complex libraries of classes. An example of this is the Mosaic library developed by TARA Systems, which is delivered with the Embedded Wizard.

The empty class is worthy of special mention. An empty class contains no members and is normally not derived from any other class. Such classes are used exclusively as root classes in developing class libraries. All classes or certain particular classes within the library may then be derived from this class in the usual manner. The definition of an empty class is limited to the name of the class, followed by a semicolon '`;`':

```
class GenericClass;
```

Normally, the definition of a class, assuming it is not an empty class, consists of one or more members. The following members may be defined:

- Instance variables → A simple variable that stores a value during the lifetime of an object, whereby each object contains its own copy of the variable. An object's variables store the state of that object, for example the color in which a shape is to be drawn on the screen. Instance variables always belong to an object (to an instance); it is not possible to define global variables.
- Instance arrays → An array is a variable that can store more than one value. The values within the array can be accessed by using an index.
- Properties → A property is an 'intelligent' variable. In a manner completely transparent to the programmer, accessing a property invokes a special onget or onset method, in which each property has its own methods. The programmer may define the logic of these methods himself in order to react to the specific aspects of read or write access to the property. For example, the programmer can have the screen contents updated automatically whenever a property is changed.
- Embedded objects → Any object may embed other objects of a different class. Each embedded object belongs to the superior object. Each superior object contains a copy of all the objects embedded within it. When an object is created at runtime, all objects embedded within it are automatically created as well. Beside embedded objects, Chora also supports so called auto objects → see "Auto objects" (chapter 4.4).
- Methods → A method is a function of an object. In calling a method, an object can be made to perform a particular action. In the above example, the `Draw()` method serves to draw an object on the screen. The method's behavior (the logic) must be defined by the programmer. Simply expressed, methods are comparable to C functions. The unusual thing about methods is that they are always called and executed within the context (scope) of an object. It is therefore not possible to define global methods.

Optionally, the `class` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:


Copyright TARA Systems GmbH
www.embedded-wizard.de

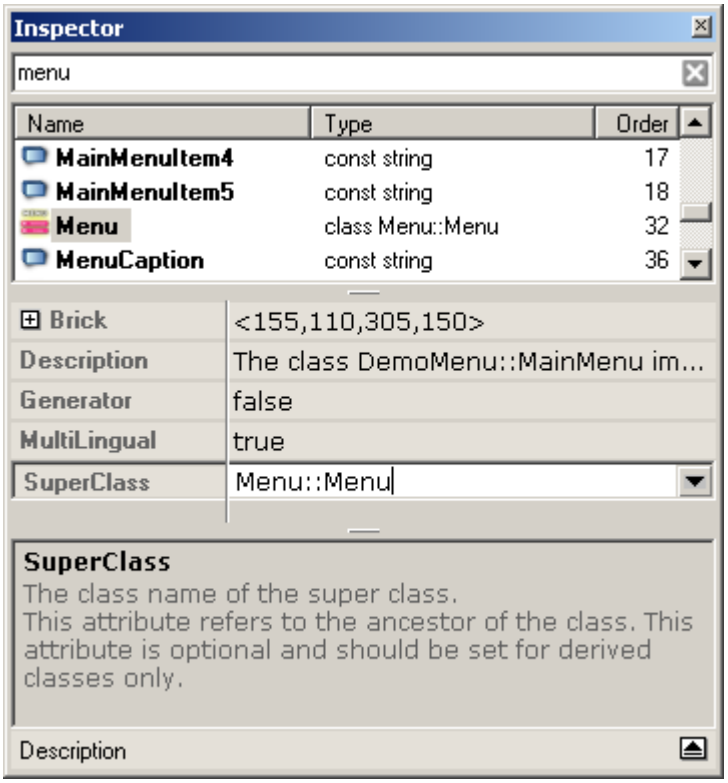
```
$output true
class Menu : Core::Group
{
  ...
}
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

In Chora it is possible to override a class by so called class variants. Class variants provide a powerful technique for the extension and adaptation of classes. Each time, the class is used, Embedded Wizard determines automatically during runtime the appropriate variant and uses it instead → for more details see "Variants" (chapter 4.8).

Within the Embedded Wizard IDE classes appear as bricks in the Unit Composer window:

A double click on the class brick will show its class members and if the class contains graphical objects, these objects are also drawn on the screen. The attributes of the class can be set directly in the Inspector window, when the appropriate class brick has been previously selected:




## 4.1.1 Instance variables

Instance variables are always introduced with the keyword `var`, followed by the data type, the name and optional an initialization value of the variable:

```
class Item
{
  var int32  Counter;
  var string Caption = "Video Settings";
  ...
}
```

In this example, the definition of the `Item` class has been given two variables: `Counter` and `Caption`. The `int32` data type of the `Counter` variable determines that the variable can store 32-bit signed integer values. The second variable, `Caption`, on the other hand, stores a `string` — that is, a sequence of characters.

A variable does not necessarily have to be initialized right when it is defined. Thus, in the example, only the `Caption` variable has been given an initial value of `"Video Settings"`. If an object of the `Item` class is created at runtime, the object's `Caption` variable is automatically given the value of the character string `"Video Settings"` at the time of creation.

Variables that are not explicitly given an initial value are assigned a default value of `0` (zero). A non-explicitly initialized `string` variable (if there is no initialization of `Caption`) would thus contain an empty string.

Chora defines a set of data types. Any of these data types may be used to define a variable. A detailed description of these data types can be found in "Data types" (chapter 6).

Access to a variable always occurs within the context of the object to which the variable belongs:

```
var Sample::Item theObject = new Sample::Item;
theObject.Caption = "Sound Settings";
```

Because each object stores its own copy of its variables, Chora does not allow the definition of global variables. Changing an object's variables has no effect on the variables of other objects:

```
var Sample::Item object1 = new Sample::Item;
var Sample::Item object2 = new Sample::Item;
theObject1.Caption = "Sound Settings";
```

If, therefore, as the above example demonstrates, the `Caption` variable of `object1` is changed, this has no effect on the contents of the second object, `object2`. Its `Caption` variable still contains the original initialization value `"Video Settings"`.

In deriving classes, all variables defined in the base class are automatically inherited by the new class. The inherited variables do not have to be redefined unless the programmer wishes to change a variable's initialization value. Then and only then, the definition of an inherited variable can be given in the derived class and initialized using a different value:

```
class SoundItem : Sample::Item
{
  inherited var Caption = "Sound Settings";
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Note the use of the keyword `inherited`. Each inherited variable that is to be re-initialized in the derived class, must be introduced using the keyword `inherited`. If this keyword is missing, the Chora compiler will report an error message. Moreover, the data type of the variable in the derived class may not be specified. Only the first definition of the variable expects the data type. Inherited variables will 'inherit' the data type from theirs ancestors.

A given variable's initialization value that is changed during derivation is valid only for objects of this derived class. Objects of any of the base classes are not affected, and their variables retain their original initialization values:

```
var Sample::Item      object1 = new Sample::Item;
var Sample::SoundItem object2 = new Sample::SoundItem;
```

In this example, the first object (an object of class `Sample::Item`) still contains the original initialization value for `Caption: "Video Settings"`. The second object, on the other hand, has been instance-modified by the derived class `Sample::SoundItem`. In this case, the object's `Caption` variable obtains the value `"Sound Settings"`.

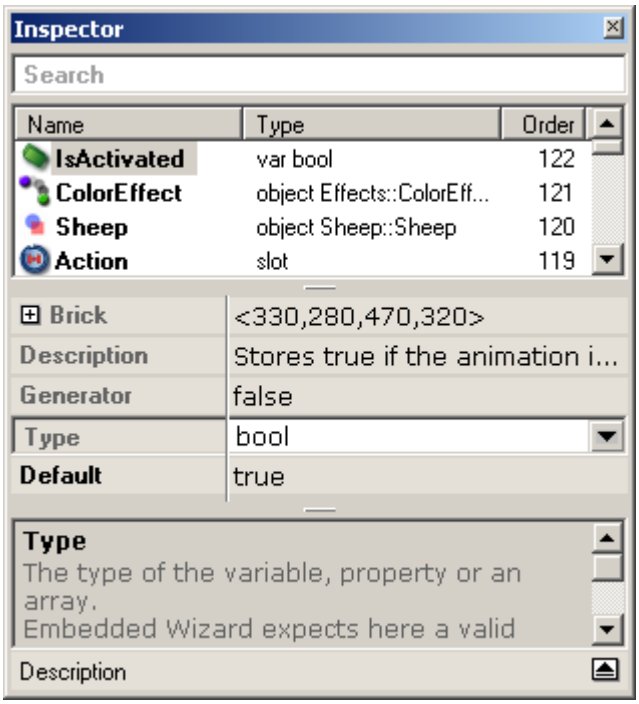
Optionally, the `var` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
var string Caption = "Hello World!";
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE variables appear as bricks in the Class Composer window:

The attributes of the variable can be set directly in the Inspector window, when the appropriate variable brick has been previously selected:




## 4.1.2 Instance arrays

Instance arrays are always introduced with the keyword `array`, followed by the data type, the name, the size and optional a set of initialization values of the array:

```
class Menu
{
  array int32  Counters[8];
  array string Captions[2] =
  (
    Default[0] = "Video Settings";
    Default[1] = "Sound Settings";
  );
  ...
}
```

In this example, the definition of the `Menu` class has been given two arrays: `Counters` and `Captions`. The data type of the `Counters` array, `int32`, determines that the array can store 32-bit signed integer values. The second array, `Captions`, on the other hand, stores only `string` values, that is, series of characters.

Unlike a variable, an array can store more than one value. To do this, the maximum size of an array must be given explicitly in its definition. In this case, the size of the two arrays was limited to `8` and `2`, respectively. The `Counters` array can thus not store more than `8` values.


Copyright TARA Systems GmbH
www.embedded-wizard.de

An array does not necessarily have to be initialized at the time of definition. In our example, only the `Captions` array was given initial values, namely `"Video Settings"` and `"Sound Settings"`. If an object of class `Menu` is created at runtime, the object's `Captions` array is automatically initialized at the time of creation, so that the first element of the array `[0]` is given the character string `"Video Settings"` and the second element `[1]` the string `"Sound Settings"`. Please note that the elements of an array are always counted from `0`. The first element is number `0`, the second `1`, and so on.

The values of an array that have not explicitly been given an initialization value are given a default value of 0 (zero). A non-explicitly initialized `string` array (if the initialization for `Captions` is missing) would thus contain only empty strings. Please note that the name `Default` must always be given at initialization and that the name does not have anything to do with the default value 0 (zero).

It is not absolutely necessary for all elements of an array to be dealt with at initialization. It is possible to explicitly initialize only selected elements of an array while the other values are preassigned the default `0` (zero) value:

```
class Menu
{
  array int32 Counters[8] =
  (
    Default[0] = 1251;
    Default[3] = 1369;
  );
    ...
}
```

As the above example demonstrates, only the first and fourth element of the array have been initialized. All other elements of the array still contain the value `0` (zero).

Chora defines a set of data types. All of these data types may be used in the definition of an array. A detailed description of data types can be found in "Data types" (chapter 6).

Access to an array always occurs in the context of the object to which the array belongs:

```
var Sample::Menu theObject = new Sample::Menu;
theObject.Captions[0] = "Brightness";
theObject.Captions[1] = "Contrast";
```

Because each object stores its own copy of its array, it is not possible to define global arrays in Chora. Changing an object's array has no effect on the arrays of other objects:

```
var Sample::Menu object1 = new Sample::Menu;
var Sample::Menu object2 = new Sample::Menu;
theObject1.Captions[0] = "Brightness";
```

Thus, if — as the above example demonstrates — at runtime the first element of the `Captions` array of `object1` is altered, this has no effect on the content of the second object, `object2`. Its `Captions` array still contains the original initialization values `"Video Settings"` and `"Sound Settings"`.


Copyright TARA Systems GmbH
www.embedded-wizard.de

When classes are derived, all arrays defined in the base class are automatically inherited by the new class. The derived arrays do not have to be redefined unless the programmer wishes to change the initialization values of an array. Then, and only then, the definition of the inherited array can be restated and initialized with different values:

```
class SubMenu : Sample::Menu
{
  inherited array Captions[] =
  (
    Default[0] = "Brightness";
    Default[1] = "Contrast";
  );
}
```

Note the use of the keyword `inherited`. Each inherited array that is to be re-initialized in the derived class must be introduced using the keyword `inherited`. If this keyword is missing, the Chora compiler will give an error message. Moreover, the data type and size of the array may not be redefined in the derived class. Only the first definition of the array expects the data type. Inherited arrays will 'inherit' the data type and the size from theirs ancestor.

The initialization values for an array that are changed during derivation are valid only for objects of the derived class. Objects of one of the base classes are not affected by the change and their arrays retain the original initialization values:

```
var Sample::Menu    object1 = new Sample::Menu;
var Sample::SubMenu object2 = new Sample::SubMenu;
```

In this example, the first object (an object of class `Sample::Menu`) still contains the original initialization values for `Captions`, namely `"Video Settings"` and `"Sound Settings"`. The second object, on the other hand, was instantiated by the derived class `Sample::SubMenu`. In this case, the object's `Captions` array will contain the two values `"Brightness"` and `"Contrast"`.

Additionally it is possible to define an array consisting of more than one dimensions. For this purpose the sizes of all dimensions have to be placed between the braces and separated by commas:

```
class Menu
{
  array int32 Field[4,5] =
  (
    Default[0,0] = 1;
    Default[0,1] = 2;
    ...
    Default[3,4] = 20;
  );
    ...
}
```

The access to an multi dimensional array works in the same way as the access to a single dimensional array. The index of the required array element consists of more expressions separated by commas:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Sample::Menu object3 = new Sample::Menu;
Object3.Field[0,2] = 1369;
```

If possible omit the usage of multi dimensional arrays because of possible changes in the future implementation of Chora. The current implementation of arrays will be then replaced in order to support dynamic and open arrays.

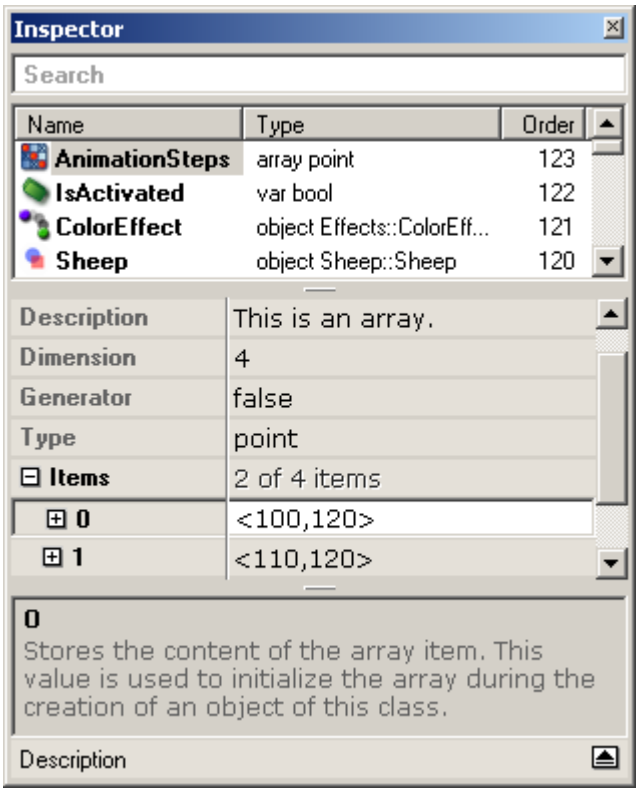
Optionally, the `array` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
array bool Field[5];
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE arrays appear as bricks in the Class Composer window:

The attributes of the array can be set directly in the Inspector window, when the appropriate array brick has been previously selected:




## 4.1.3 Properties

Properties are always introduced with the keyword `property`, followed by the data type, the name and optional an initialization value of the property:

```
class Item
{
  property int32  Counter;
  property string Caption = "Video Settings";
  ...
}
```

In this example, the definition of the `Item` class has been given two properties: `Counter` and `Caption`. The data type of the `Counter` property `int32` specifies that the property can store 32-bit signed integer values. The second property, `Caption`, on the other hand, stores a `string` — that is, a sequence of characters.

A property does not necessarily have to be initialized right when it is defined. In the example, only the `Caption` property has been preassigned an initialization value, namely `"Video Settings"`. Thus if at runtime an object of the `Item` class is created, the object's `Caption` property is automatically preassigned the character string `"Video Settings"`.

Properties that have not been explicitly given a value are assigned a default value of 0 (zero). A `string` property that has not been explicitly initialized (if the initialization of `Caption` is missing) would thus contain an empty string.

Chora defines a set of data types. All of these data types may be used in defining a property. A detailed description of the data types is found in "Data types" (chapter 6).

Access to a property always occurs in the context of the object to which the property belongs:

```
var Sample::Item theObject = new Sample::Item;
theObject.Caption = "Sound Settings";
```

From the programmer's perspective, a property can be compared with a variable → properties are handled the same as simple variables. The difference between variables and properties lies in the strict use of the onget and onset methods. Each property must define one onget and one onset method. The logic of these methods is automatically executed when the property's value is read or when a new value is assigned to the property. The programmer uses the onget and onset methods to handle access to the property in a particular way. For example, he can react to changes to the `Caption` property by having the screen refreshed:


Copyright TARA Systems GmbH

```
class Item
{
  property string Caption = "Video Settings";

  onget Caption
  {
    return pure Caption;
  }

  onset Caption
  {
    pure Caption = value;

    // And now redraw the screen to display new
    // Caption
    UpdateScreen();
  }
}
```

In spite of the fact that any access to the property occurs by means of the onget or onset method, each property (just as with a variable) has its own area in memory in which the property's value can be kept. In order to access this area of memory, the keyword `pure` must be used. With this keyword, the property's onget and onset methods are circumvented and the property's value is read directly from memory or changed in memory, as with a simple variable.

The above example demonstrates the use of the keyword `pure` within the onget and onset methods. The onget method returns the value from the property's area of memory. The onset method changes the content of this memory and forces a screen refresh.

It is not mandatory that the value of a property is placed in memory. In many uses, the onget and onset methods determine the property's value on request — for example, in the case of a property that returns the current time. Storing the value would not make sense in such a case. Further details on the definition and use of the onget and onset methods are found in "onget methods" (chapter 4.1.5.2), "onset methods" (chapter 4.1.5.3) and in "Reference data types" (chapter 6.3).

Because each object stores its own copy of its properties, it is not possible to define global properties in Chora. Changing a property of an object does not (usually) have any effect on the properties of other objects:

```
var Sample::Item object1 = new Sample::Item;
var Sample::Item object2 = new Sample::Item;
theObject1.Caption = "Sound Settings";
```

Thus if, as the above example demonstrates, the `Caption` property of `object1` is changed at runtime, this does not usually have an effect on the content of the second object, `object2`. Its `Caption` property still contains the original initialization value `"Video Settings"`, provided that the property's onget and onset methods are not dependent on each other. The programmer can namely query values of other objects using the onget methods, and change other properties, variables or objects using onset methods. Thus, simple access to a property can have an affect on other variables, properties or objects.


Copyright TARA Systems GmbH

When classes are derived, all properties defined in the base class, including their onget and onset methods, are automatically inherited by the new class. The inherited properties do not have to be redefined unless the programmer wishes to change the initialization value of a property. Then and only then, the definition of the inherited property can be restated in the derived class and initialized using a different value:

```
class SoundItem : Sample::Item
{
  inherited property Caption = "Sound Settings";
}
```

Note the use of the keyword `inherited`. Any inherited property that is to be re-initialized in the derived class must be introduced using the keyword `inherited`. If this keyword is missing, the Chora compiler will give an error message. Moreover, the data type of the property in the derived class may not be specified. Only the first definition of the property expects the data type. Inherited properties will 'inherit' the data type from theirs ancestors.

A property's initialization value, changed during derivation, is valid only for objects of the derived class. Objects of one of the base classes are not affected by the change, and their properties retain their original initialization values:

```
var Sample::Item      object1 = new Sample::Item;
var Sample::SoundItem object2 = new Sample::SoundItem;
```

In this example, the first object (an object of class `Sample::Item`) still has the original initialization value for `Caption`, namely `"Video Settings"`. The second object, on the other hand, has been initialized by the derived class `Sample::SoundItem`. In this case, the object's `Caption` property will be given the value `"Sound Settings"`.

Unlike with variables, a property's initialization process requires additional explanation in order for one to understand the underlying mechanisms. If a property is initialized first time (without an `inherited` keyword), the initialization value used in the definition of the property is copied directly into the property's memory. There is no call to the onset method. This is exactly the same as when a simple variable is initialized:

```
class Item
{
  // Copy the string "Video Settings" to the
  // memory area of the property (like pure
  // keyword)
  property string Caption = "Video Settings";
  ...
}
```

If, on the other hand, a property that has already been defined in the base class is re-initialized in the derived class, then the onset method will be called, so that the base class is notified of the change:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
class SoundItem : Sample::Item
{
  // Calls the base class version of the properties
  // own onset method and passes "Sound Settings"
  inherited property Caption = "Sound Settings";
}
```

When a property is re-initialized, it is always the base version of the onset method that is invoked. Even if the onset method has been explicitly overridden in the derived class, the method version originally defined in the base class is the one invoked. Only when the object's initialization is finished are the overridden methods activated.

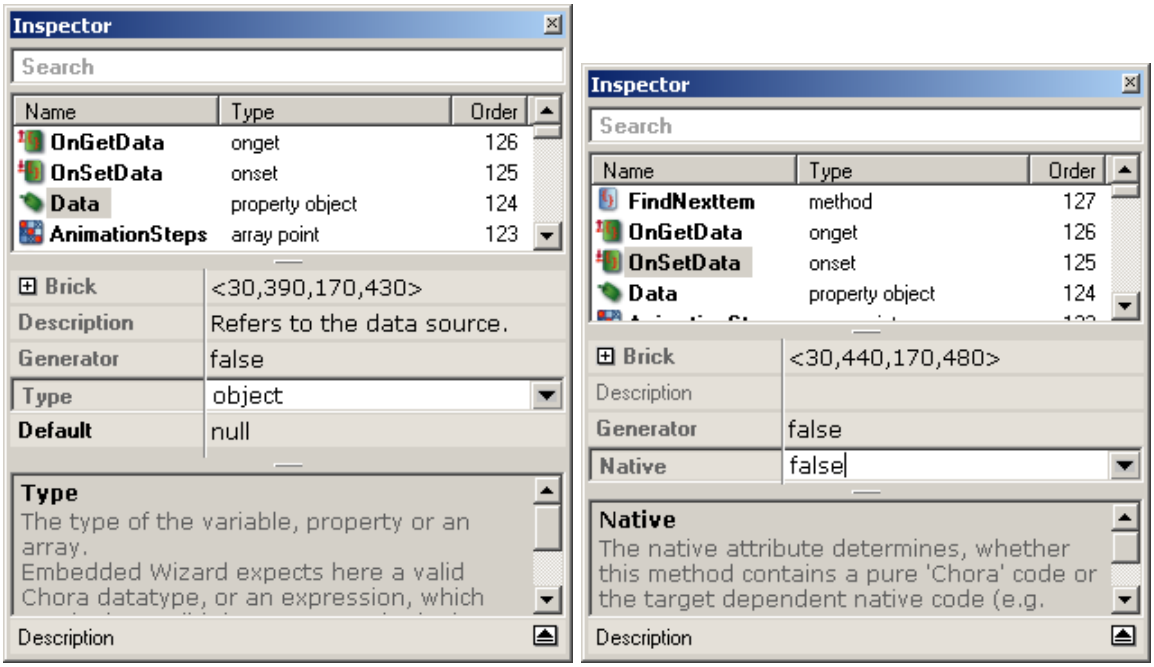
Optionally, the `property` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
property string Caption = "Hello World!";
```

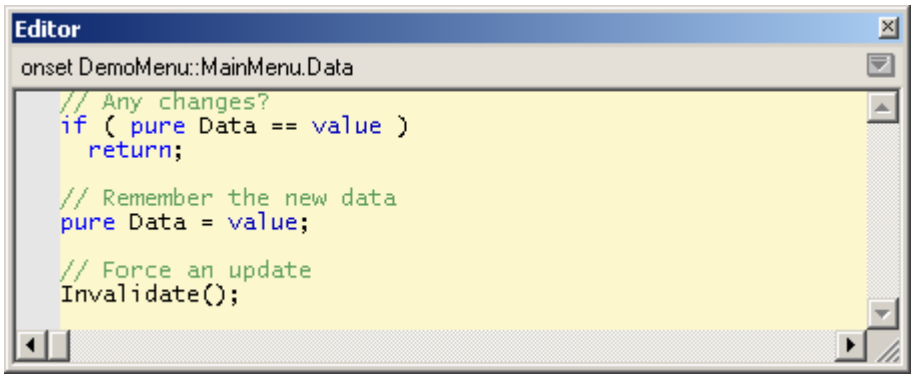
For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE properties and their onset/onget methods appear as bricks in the Class Composer window:

The attributes of the property and their onset/onget method can be set directly in the Inspector window, when the appropriate brick has been previously selected:




The logic of the onset/onget method can be implemented in the Editor window, after the method brick has been double clicked:




## 4.1.4 Embedded objects

Embedded objects are always introduced with the keyword `object`, followed by the objects class, the name and optional a set of initialization values (presets) of the object:

```
class Menu
{
  object Sample::Item Item1
  {
    preset Caption = "Video Settings";
    preset Background = #FF2020FF;
  }
  ...
}
```

In this example, the definition of the `Menu` class has been given an embedded object named `Item1`. The embedded object is an instance of the class `Sample::Item`. The object's two properties, `Caption` and `Background`, are immediately initialized with new values. This definition results in each `Menu` object having its own embedded `Sample::Item` object.

The `preset` keywords force the properties of an object to be re-initialized. If a property is not explicitly re-initialized using `preset`, the property retains the value originally defined in the object's class. Thus only properties that have been re-initialized using `preset`, are given new values. When the properties are re-initialized, the corresponding onset methods are automatically invoked, so that the object can react to the change in properties. Note that the name of the re-initialized property must fit to one of the properties defined in the object's class. If this is not the case the Chora compiler reports an error message.

An embedded object does not necessarily have to be initialized at the time of definition. Often it suffices to use the property values defined in the object's class:

```
class Menu
{
  object Sample::Item Item1;
  ...
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

In this case, all properties of the object retain their original value. No properties are re-initialized.

The embedded object can be accessed by using its name, in order to change the value of one of its properties, for example, or by using one of its methods:

```
var Sample::Menu theObject = new Sample::Menu;
theObject.Item1.Caption = "Sound Settings";
theObject.Item1.DoSomething();
```

When classes are derived, all objects defined in the base class are automatically inherited by the new class. The inherited objects do not have to be redefined unless the programmer wishes to change the initialization value of the one of the properties of the inherited objects. Then and only then, the definition of the inherited objects in the derived class can be restated and initialized using a different value:

```
class SoundMenu : Sample::Menu
{
  inherited object Item1
  {
    preset Caption =  "Sound Settings";
  }
}
```

Note the use of the keyword `inherited`. Any inherited object that is to be re-initialized in the derived class must be introduced using the keyword `inherited`. If this keyword is missing, the Chora compiler will give an error message. Moreover, the class name of the object in the derived class may not be specified. Only the first definition of the object expects the class name. Inherited objects will 'inherit' the class name from theirs ancestors.

An embedded object's initialization value, changed in the derivation, is valid only for objects of this derived class. Objects of one of the base classes are not affected by the change, and their embedded objects retain their original initialization values:

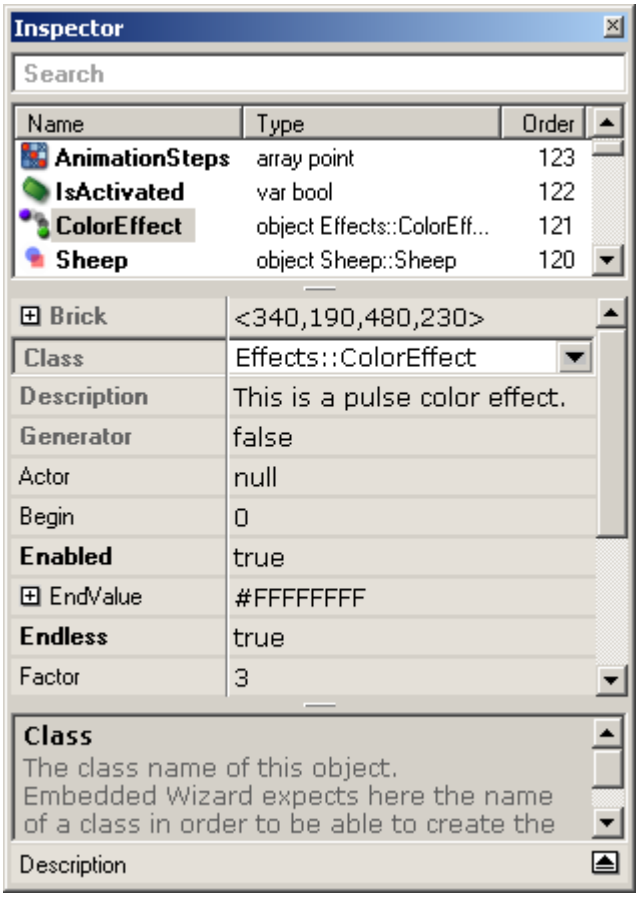
```
var Sample::Menu      object1 = new Sample::Menu;
var Sample::SoundMenu object2 = new Sample::SoundMenu;
```

In this example, the first object (an object of class `Sample::Menu`) still contains the original initialization value for its embedded object, `Item1`. The second object, on the other hand, has been instantiated by the derived class `Sample::SoundMenu`. In this case, the `Caption` property of the embedded object will contain the value `"Sound Settings"`.

Within the Embedded Wizard IDE non-graphical objects appear as bricks in the Class Composer window:

In case of graphical objects, the objects appear as this is done on the screen in the target system: the filled rectangle object is drawn as a filled rectangle, a menu item object appears as a menu item, etc.

The attributes of the object can be set directly in the Inspector window, when the appropriate object's brick has been previously selected:




## 4.1.5 Methods

Methods are always introduced with the keyword `method`, followed by the return data type, the method name, argument list and the logic (the body) of the method:

```
class Menu
{
  method void HandleFocus( arg bool aObtain )
  {
    if ( aObtain )
      // Do something in response to the obtained focus,
      // etc.
  }
}
```

This example shows the definition of a class `Menu` with a single method, `HandleFocus()`. The definition of the method determines precisely how it is to be used, when it is called. Thus, this method expects nothing other than one argument of type `bool`. The method's return type has been given as `void`, meaning the method can't return a value. The Chora language offers various data types that you can use in defining a method. You will find further details in "Data types" (chapter 6).

The method's logic is enclosed by two braces `{}`. Here the programmer defines which statements are to be executed when the method is invoked. He can use conditions, loops, local variables, expressions, etc. You'll find further details in "Statements" (chapter 5).

Calling a method always occurs in the context of the object to which the method belongs. If the method expects arguments, the arguments must be enclosed by parentheses when they are passed on at the time the method is called:

```
var Sample::Menu theObject = new Sample::Menu;
theObject.HandleFocus( true );
```

Thus, the above example invokes the `HandleFocus()` method in the context of the object `theObject`. In accordance with the definition of the `HandleFocus()` method, this method expects exactly one `bool` argument. In this case the method is invoked with the value `true` as argument.

Calling a method in the context of an object means that the members of the object are directly accessible from the method's logic. Thus, for example, the value of the variable `isFocused` can be accessed directly from the method:

```
class Menu
{
  var bool isFocused = false;

  method void HandleFocus( arg bool aObtain )
  {
    isFocused = aObtain;
    ...
    // Do something to handle the focus change.
  }
}
```

Please note that each object contains its own copy of all variables, properties and arrays that belong to it. Thus, a method can access only those variables in the context of the object of which it has been invoked. Variables of other objects can't be accessed directly. This is the main difference between methods and the functions familiar from the C programming language.

If desired, a method can also expect more than one argument, separated by commas. Each argument must begin with the keyword `arg`:

```
class Shape
{
  method void DrawLine( arg point aStart,
    arg point aEnd )
  {
     // Do something to draw a line from aStart
     // to aEnd point.
     ...
  }
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

If a method expects more than one argument, all the arguments must be separated by commas at the time the method is called. The following example demonstrates how the method `DrawLine()` has been called, with two points `<10,20>` and `<100,120>` as arguments:

```
var Sample::Shape shape = new Sample::Shape;
shape.DrawLine( <10,20>, <100,120> );
```

A method does not necessarily always have to expect arguments. In fact, it often makes sense to define methods without arguments. In such cases, the list of arguments is omitted in the definition and when the method is called. What remain are the parentheses:

```
class shape
{
  method void Show()
  {
    // Do something to show the graphical object.
  }
}
```

If a method is called with the wrong number of arguments, or the data type of arguments given do not match the definition of the method anywhere, the Chora compiler will report an error message and the translation will be aborted.

If the return value of a method is not `void`, the method must always return a value. The appropriate statement `return` then terminates the method and returns value to the caller:

```
class Math
{
  method int32 Power2( arg int32 aValue )
  {
    return aValue * aValue;
  }
}
```

As this example indicates, the `Power2()` method returns an `int32` (32-bit signed integer) value. In the logic of the method, the argument `aValue` is used to calculate the power$^2$ value, and this result is returned. The caller of the `Power2()` method can then use this result directly in an expression:

```
var Sample::Math math = new Sample::Math;
var int32        x    = 10;
var int32        y    = 32;
var int32        result;

// result = x^2 + y^2
result = math.Power2( x ) + math.Power2( y );
```

Through class derivation, all methods defined in the base class are automatically inherited by the new class. The inherited methods do not have to be redefined unless the programmer wishes to change the behavior of the inherited method. Then and only then, the definition of the inherited method may be declared again and assigned a different logic:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
class SubMenu : Sample::Menu
{
  inherited method HandleFocus()
  {
    // If the menu loses the focus, hide this menu
    if ( aObtain == false )
      Hide();

    // Let the super class a chance to respond to the
    // focus change.
    super( aObtain );
  }
}
```

Note the use of the keyword `inherited`. Each inherited method that is to be used anew in the derived class must be introduced using the keyword `inherited`. If this keyword is missing, the Chora compiler will report an error message. Moreover, the return data type and the arguments list in the derived class may not be specified. Only the first definition of the method expects the return data type and optionally the arguments list. Inherited methods will 'inherit' the return data type and the arguments list from theirs ancestors.

The implementation of the methods, changed in the derivation, is valid only for objects of this derived class. Objects of one of the base classes are not affected by the change, and their methods retain the originally defined logic.

The derivation and overriding of methods form one of Chora's most important concepts. It is a simple matter for the programmer to extend the behavior of a derived class without having to change the definition of this class. He needs only to override all the methods that are affected by the change. All other methods that are not explicitly overridden are automatically inherited by the derived class.

Please note that the inheriting of methods does not at all mean that there will be more than one copy of one and the same method at runtime. As long as a method has only been inherited and not overridden, the base class and the inherited classes share this single implementation of the method.

Overriding a method allows the programmer to change a method's behavior completely. If such an overridden method is called at runtime, the logic of this method is run. The method originally defined in the base class is no longer executed – the base version of the method is overridden.

Often, however, it makes sense for the overridden method in the derived class to handle only one specific case and to direct all other cases to the method's original implementation in the base class. The `HandleFocus()` method demonstrates this kind of use. The method's original version was defined in the `Sample::Menu` class. The `SubMenu` class overrides only this method and extends it to include handling of the focus loss (`aObtain == false`). In all other cases the original version of the method can be used. The method is called using the super call `super()`. This technique prevents the logic of a method from being copied without reason.

The Chora programming language categorizes methods according to their area of use:


Copyright TARA Systems GmbH

- native method → Allows Chora code to be mixed with the code of the target system. In the native method, you can directly adopt C, C++, assembler and other code. See also "'native' statement" (chapter 5.17).
- onget method → A specialized method that is invoked whenever the value of a property in an expression is to be queried. Each property has its own onget method. The return value of the onget method yields the queried value of the property.
- onset methods → A specialized method that is automatically invoked whenever the value of a property in an expression is to be changed. Each property has its own onset method. Using an onset method, the programmer has the means to react to the changing of a property.
- slot methods → A specialized method that can handle signals from other objects. Signals exist for the sake of simple communication between objects. If an object sends a signal to a particular slot of another object, the logic of the corresponding slot method of the recipient object is automatically executed.
- Init constructors → Constructors are special methods that are automatically invoked in order to complete the initialization of an object. Usually, objects are initialized only by assigning values to the object's variables and properties. In rare cases it may, however, be necessary for an object to execute additional code during its initialization. For example, thanks to a constructor, each newly created timer object is automatically registered by the operating system in order for it to be able to receive the timer events.
- Done destructors → Destructors are special methods that are automatically invoked shortly before an object is destroyed, in order to give the object the chance to release system resources that it is no longer using. The Chora programming language has a Garbage Collector that automatically handles the disposal of unused (garbage) objects; but in rare cases, when an object has itself reserved a system resource, this resource must be properly released when the object is destroyed. For example, the timer object must be dismissed shortly before its destruction by the operating system in order not to receive any more timer events. In this case, the timer object's destructor handles the required deregistration from the operating system.
- ReInit re-constructors → Re-constructors are special methods that are automatically invoked in order to complete the re-initialization of an object. The re-initialization is triggered by the language selection. Each time, a preferred language is selected by the user, the Chora Runtime Environment forces all existing objects to adapt themselves to this new language. In this manner language dependent constants and resources are reloaded into the objects and the GUI application appears in the newly selected language on the screen automatically. In rare cases it may, however, be necessary for an object to execute additional code during its re-initialization. This can be done in this ReInit method.

  Note, unlike the Init constructor, the re-constructor is called multiple times during the lifetime of an object.

Optionally, the method definitions may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

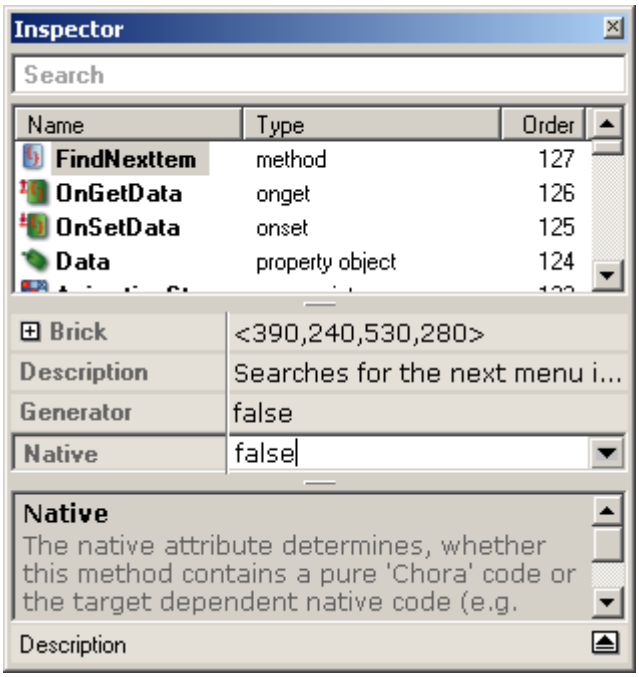

Copyright TARA Systems GmbH

```
$output true
method void LoadAudioSetting( bool aMode )
{
  ...
}
```

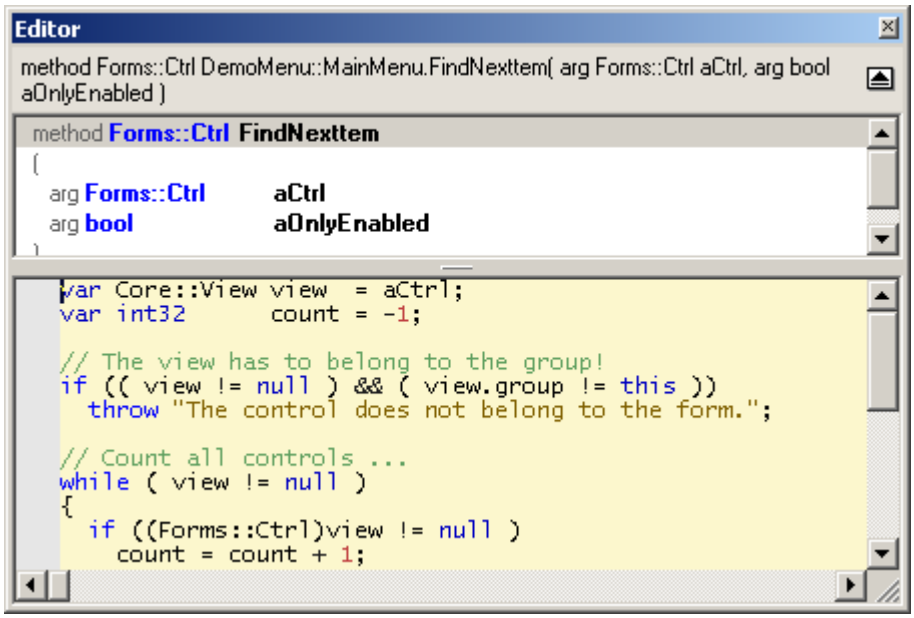
For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE the methods appear as bricks in the Class Composer window. The icon in the brick depends on the kind of the method. In case of generic methods following brick is displayed:

The attributes of the method can be set directly in the Inspector window, when the appropriate method's brick has been previously selected:




The declaration and the logic of the method can be implemented in the Editor window, after the method brick has been double clicked:


Editor
method Forms::Ctrl DemoMenu::MainMenu.FindNextItem( arg Forms::Ctrl aCtrl, arg bool aOnlyEnabled )
method Forms::Ctrl FindNextItem
(
arg Forms::Ctrl aCtrl
arg bool aOnlyEnabled
)
var Core::View view = aCtrl;
var int32 count = -1;
// The view has to belong to the group!
if (( view != null ) && ( view.group != this ))
throw "The control does not belong to the form.";
// Count all controls ...
while ( view != null )
{
if ((Forms::Ctrl)view != null )
count = count + 1;

### 4.1.5.1 Native methods

Normally the logic of a method is defined in the Chora language, and the Chora compiler is responsible for translating the logic into statements appropriate to the target system. Unfortunately, Chora does not permit direct access to the hardware or to the services of the operating system. This limitation is, however, a reasonable one, given that in the GUI developed in Chora aims for the greatest possible degree of platform-independence. If direct access to the hardware or operating system is nonetheless necessary, it can be done within a native method in the language of the target system (e.g. ANSI C).

The use of native methods allows a mixture of Chora and native code. However, it prevents platform-independent GUI development. In practice, native methods should be used only in developing driver classes. A driver class serves as an interface between the GUI and the underlying target system. By adapting the driver class, the GUI program can be ported to other target platforms.

The definition of a native method is specially signed using the keyword `native`. During translation, the Chora compiler ignores the logic of a native method and places it unchanged in the generated code. Thus, one is in a position to mix arbitrary C, C++ or assembler code with Chora, and to define a clear interface between the GUI application and the target:

```
class AudioDevice
{
  method native void Mute()
  {
    $if $platform == Tara.Win32.*
        __send_i2c( Audio, OFF );
    $endif
  }
}
```

In this example, a class `AudioDevice` has been defined with a native method `Mute()`. This method can, like any other method, be called directly in Chora. The main difference between this and a normal method is that its logic was written in the language of the target system (in this case, ANSI C). Thus the native method invokes a fictitious `__send_i2c()` function in order to, for example, turn off the loudspeaker in a television set.

A native method can, like any other method, expect arguments, and deliver return values. Within the logic of a native method, it is even possible to access objects, call other methods, etc. All of this happens naturally in the language of the corresponding target system, and requires a knowledge of the internal functioning of Chora objects and the Runtime Environment. We recommend examining the code generated by the Chora compiler if you wish to write complex native methods.

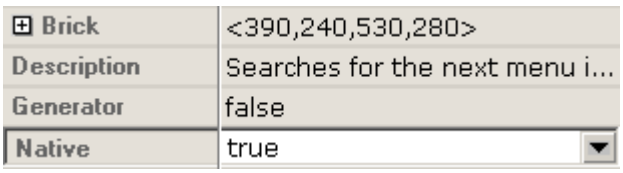
In order to differentiate between different target platforms within a native method definition, it is necessary to use the Chora preprocessor directives:

```
class AudioDevice
{
  method native void Mute()
  {
    $if $profile == Chassis1
       __send_i2c( Audio, OFF );
    $endif
    $if $profile == Chassis2
       AudioOff();
    $endif
  }
}
```

By using the `$if`, `$else`, `$elseif` and `$endif` Chora preprocessor directives, part of the code block, depending on the target platform, can be ignored or taken over. If the condition in an `$if` or `$elseif` directive is fulfilled (is `true`), the following part of the code block is taken over. If the condition is not fulfilled, the code block is ignored. Details of the Chora preprocessor are found in "Chora preprocessor" (chapter 8).

Beside the native methods, a more powerful `native` statement is available in Chora. The statement allows the mixing of Chora code and the target specific code within the body of one and the same method. For more details see "'native' statement" (chapter 5.17).

Within the Embedded Wizard IDE the native method attribute can be set in the Inspector window, when the appropriate method has been previously selected:




### 4.1.5.2 onget methods

onget Methods are always introduced with the keyword `onget`, followed by the name of the corresponding property and the logic (the body) of the method:

```
class Menu
{
  property color Color = #000000FF;

  onget Color
  {
    // Get the value from memory
    return pure Color;
  }
}
```

In the above example, the `Menu` class has been given a property named `Color` as well as a corresponding onget method. Note that the name of the property must correspond exactly to the definition of the `onget` method. If this is not the case, the Chora compiler will report an error message. Unlike other methods, `onget` doesn't expect the return data type nor the arguments list. The Chora compiler will adopt the data type of the property as the return data type of the corresponding `onget` method automatically.

Each time the value of the `Color` property is queried, the property's onget method is called automatically:

```
var Sample::Menu theObject = new Sample::Menu;
var color        theColor;

theColor = theObject.Color;
```

The onget method's returned value thus determines the value of the property evaluated in an expression. In the above example, the onget method gives the true value of the property by reading it directly from memory, using the keyword `pure`. Often, the onget method contains specialized logic to determine the value of the queried property on request: for example, a property that must always return the current time when queried.

Except for the definition that uses the keyword `onget`, an onget method is no different from a normal method. All onget methods are automatically inherited in class derivation and may be overridden in the derived class:

```
class SubMenu : Sample::Menu
{
  inherited onget Color
  {
    return ... // Another value, etc.
  }
}
```

Additionally, an onget method can, like any other method, contain native code:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
class Time
{
  property int32 Seconds = 0;

  onget native Seconds
  {
    $if $platform == Tara.Win32.*
      return clock();
    $endif
  }
}
```

Further details on properties are found in "Properties" (chapter 4.1.3) and "Reference data types" (chapter 6.3).

#### 4.1.5.3 onset methods

onset Methods are always introduced with the keyword `onset`, followed by the data type of the corresponding property, the properties name and the logic (the body) of the method:

```
class Menu
{
  property color Color = #000000FF;

  onset Color
  {
    // Store the new value in the memory
    pure Color = value;
  }
}
```

In the above example, the `Menu` class has been given a property named `Color` and a corresponding onset method. Note that the name of the property must correspond exactly to the definition of the `onset` method. If this is not the case, the Chora compiler will report an error message. Unlike other methods, `onset` doesn't expect the return data type nor the arguments list. The Chora compiler will adopt the data type of the property for the hidden `value` argument of the corresponding `onset` method automatically.

Each time that the value of the `Color` property is to be changed in an expression, the property's onset method is called automatically:

```
var Sample::Menu theObject = new Sample::Menu;

theObject.Color = #FF00FFFF;
```

The assignment to the `Color` property shown in the example automatically invokes the corresponding onset method of the `Color` property. The assigned value is passed on to the method in a hidden argument named `value`. Within the onset method, you can evaluate `value` and change it in order to react to changes in the property. You can, for example, query the validity of the assigned value and, if necessary, adjust the invalid value:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
class Menu
{
  property color Color = #000000FF;

  onset Color
  {
    // No transparency allowed!
    if ( value.alpha < 255 )
      value.alpha = 255;

    // Store the new value in the memory
    pure Color = value;
  }
}
```

If the property's new value is to be stored, you must place the contents of `value` directly in memory, using the keyword `pure`. The above example demonstrates the usage of `pure` keyword and the hidden `value` argument.

An onset method can't return any values.

Except for its definition using the keyword `onset`, there is no difference between an onset method and a normal method. All onset methods are automatically inherited in class derivation and can be overridden in the derived class:

```
class SubMenu : Sample::Menu
{
  inherited onset Color
  {
    // Pass the value to the base class version of
    // the onset Method.
    super( value );

    // After the value has been stored in the base
    // class version of the onset method, force
    // an update of the screen, etc.
    UpdateAll();
  }
}
```

In addition, an onset method may contain native code, just as with any other method:

```
class Time
{
  property int32 Seconds = 0;

  onset native Seconds
  {
    $if $platform == Tara.Win32.*
      setclock( value );
    $endif
  }
}
```


Copyright TARA Systems GmbH

Further details on properties are found in "Properties" (chapter 4.1.3) and "Reference data types" (chapter 6.3).

#### 4.1.5.4 slot methods

slot Methods are always introduced with the keyword `slot`, followed by the slots name and the logic (the body) of the method:

```
class Menu
{
  slot OnAction
  {
    // Do something in response to the signal ...
  }
}
```

In the above example, the `Menu` class has been given a slot method, `OnAction`. Note that a slot method defines neither arguments nor return data types. Only the name of the slot method is specified. When referring to the handling of signals, a slot method is known simply as a slot.

The logic of the slot method is automatically executed whenever a signal arrives at the slot. The process of sending a signal occurs through the use of the `signal` statement. The following example demonstrates how a sender object sends a signal to the `OnAction` slot of object `theObject`:

```
var Sample::Menu theObject = new Sample::Menu;

signal theObject.OnAction;
```

Signals and slot methods exist for simple communication. A receiver object is informed that an event has happened. The transmission of additional data is not necessary. Only the identity of the sender object can be evaluated. For this purpose, each slot method contains a hidden argument named `sender` through which access to the broadcast object can be obtained. If a slot method is to receive signals from different sources, `sender` can be used to verify the source of a signal:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
class Menu
{
  // Two embedded objects which can send signals
  // to the OnAction slot.
  object Sample::Item Item1;
  object Sample::Item Item2;

  slot OnAction
  {
    if ( sender == Item1 )
      // Do something in response to the signal from
      // Item1 object;

    if ( sender == Item2 )
      // Do something in response to the signal from
      // Item2 object;
  }
}
```

Except for the definition using the keyword `slot`, there is no difference between a slot method and a normal method. All slot methods are automatically inherited in class derivation and can be overridden in the derived class:

```
class SubMenu : Sample::Menu
{
  object Sample::Item Item;

  inherited slot OnAction
  {
    if ( sender == Item )
      // Do something in response to the signal
      // from the Item object ...
    else
      // or call the base class version of the slot
      // method to handle signals from other objects.
      super( sender );
  }
}
```

In addition, a slot method can, like any other method, contain native code:

```
class Device
{
  slot native OnOpenTray
  {
    $if $profile == Chassis1
      OpenTray();
    $endif
  }
}
```

Note that in addition to the keyword `slot`, there is an identically named instant data type called `slot`. This data type exists to define variables in which the entry point of a slot method can be stored:

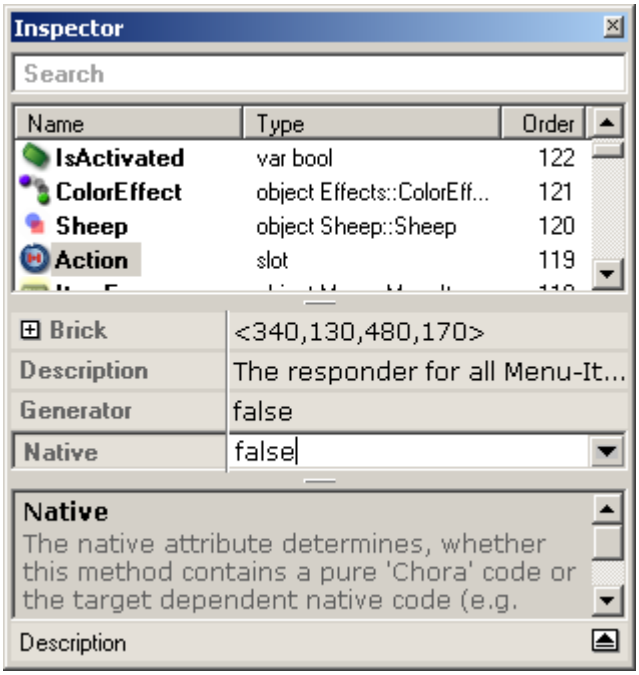
```
var slot theSlot = theObject.OnAction;

// send a signal to the slot stored in the variable
// theSlot.
signal theSlot;
```

The use of slot variables simplifies the insulation between the sender and receiver objects. The sender object is no longer dependent on knowing the identity of the receiver object. It simply sends a signal without having to worry about which object will be receiving that signal. At runtime, the programmer can initialize the variable with the entry point of another slot; in this process, the signals are sent to a different slot without the sender object being having to be notified.

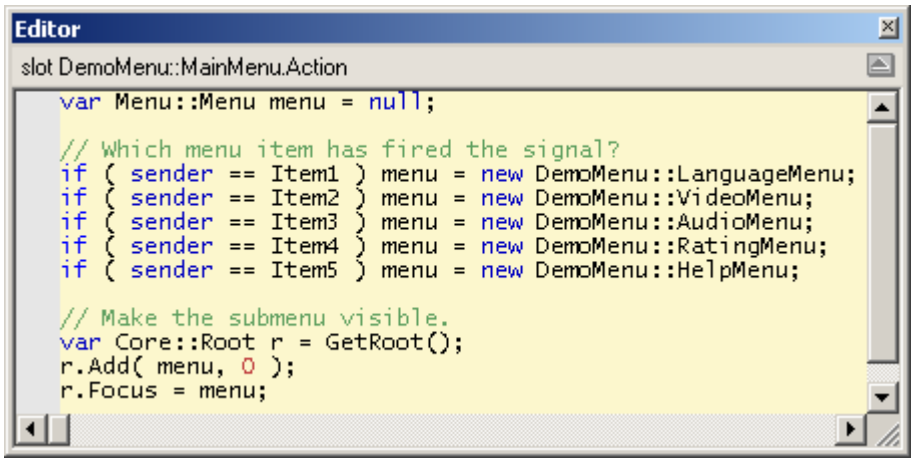
Additionally, the statement `postsignal` and `idlesignal` can be used to send a pending signal after a short delay. Further details on signals can be found in "'signal' statement" (chapter 5.11), "'postsignal' statement" (chapter 5.12) and "'idlesignal' statement" (chapter 5.13)

Within the Embedded Wizard IDE the slot methods appear as bricks in the Class Composer window:

The attributes of the slot method can be set directly in the Inspector window, when the appropriate method's brick has been previously selected:




The logic of the method can be implemented in the Editor window, after the method brick has been double clicked:


Editor
slot DemoMenu::MainMenu.Action
var Menu::Menu menu = null;
// Which menu item has fired the signal?
if ( sender == Item1 ) menu = new DemoMenu::LanguageMenu;
if ( sender == Item2 ) menu = new DemoMenu::VideoMenu;
if ( sender == Item3 ) menu = new DemoMenu::AudioMenu;
if ( sender == Item4 ) menu = new DemoMenu::RatingMenu;
if ( sender == Item5 ) menu = new DemoMenu::HelpMenu;
// Make the submenu visible.
var Core::Root r = GetRoot();
r.Add( menu, 0 );
r.Focus = menu;

#### 4.1.5.5 Init constructors

The definition of a constructor is similar to an ordinary method. A constructor always has the name `Init`, expects none or exactly one argument of data type `uint32` (unsigned 32-bit integer) and may not return a value:

```
class Timer
{
  method native void Init()
  {
    $if $platform == Tara.Win32.*
      // Allocate an OS timer, etc.
      StartTimer( ... );
    $endif
  }
}
```

Note, the constructor may declare a single `uint32` argument. This optional argument is for internal purpose only. Usually, the user-defined constructors should never use this optional argument:

```
class Timer
{
  method native void Init( arg uint32 aArg )
  {
    $if $platform == Tara.Win32.*
      // Allocate an OS timer, etc.
      StartTimer( ... );
    $endif
  }
}
```

The constructors are automatically called by the Chora Runtime Environment when an object is created, to give the object the chance to request additional resources, etc. It is not necessary and, in fact, unadvisable to call the constructors directly from the Chora code! This can lead to inexplicable runtime errors:


Copyright TARA Systems GmbH

```
var Core::Timer timer = new Core::Timer;

// DON'T call constructors directly
timer.Init(0);
```

Because constructors are optional, the absence of constructors does not lead to an error message; they are simply ignored. The Chora compiler recognizes whether a class has defined a constructor or not, and generates the relevant initialization code, in order to correctly initialize the objects of this class.

Constructors are used almost exclusively with driver classes that require direct access to the operating system and hardware. In this case, constructors ensure correct hardware initialization and access to the operating system while the object is being created. Classes that do not require access to the hardware or the operating system should not define constructors.

Except for the above-mentioned restrictions on the definition of a constructor, there are no further differences between constructors and methods. Thus, constructors are inheritable in the derivation of classes and can be overridden in the derived classes:

```
class SuperTimer : Core::Timer
{
  inherited method Init()
  {
    // Do some stuff here, but DON'T make super call!!!
    // super( aArg )
  }
}
```

It must merely be noted that the Runtime Environment automatically calls all of an object's constructors when an object is created. This process begins with the object's oldest base class and continues during initialization along the hierarchy of inheritance.

It can be inferred from the above example of the `SuperTimer` class that `SuperTimer` was derived from `Core::Timer`. Thus, before the `Init()` method of the `SuperTimer` class is executed, the Runtime Environment takes care of initializing the `Core::Timer` class and calling its `Init()` method. Only after that is the constructor of `SuperTimer` called. Please note that this process is automatically controlled by the Runtime Environment, so that the use of the super call `super()` is superfluous and even undesirable.

Use of the optional keyword `native` allows the constructors to be implemented in the form of native methods. Since constructors are primarily intended for direct access to the hardware and operating system, most constructors will thus be native methods. See also "'native' statement" (chapter 5.17).

The 32-bit unsigned integer argument is used by the Runtime Environment and the Resource Manager whenever a resource is to be loaded into a resource object. In this case, the Resource Manager passes to the object the address of the resource that is to be loaded. In all other cases, including those involving objects created directly using the Chora keyword `new`, the constructors are always called with 0 (zero) as an argument. Unlike other methods, the argument in the constructor is optional and may be omitted.


Copyright TARA Systems GmbH
www.embedded-wizard.de

Please note that the definition of constructors given here may have to be amended for new platforms.

For more details see "'new' operator" (chapter 7.6).

#### 4.1.5.6 Done destructors

A destructor is a special kind of method with the preset name `Done`. The destructor expects no arguments, nor may it return a value:

```
class Timer
{
  method native void Done()
  {
    $if $platform == Tara.Win32.*
      // release the timer
      KillTimer( ... );
    $endif
  }
}
```

Destructors are invoked automatically by the Chora Runtime Environment shortly before an object is destroyed, in order to give the object the chance to release resources it is using. It is unnecessary and even undesirable to call destructors directly from within the Chora code! This can lead to inexplicable runtime errors:

```
var Core::Timer timer = new Core::Timer;

// DON'T call destructors directly!!!
timer.Done();
```

It must be noted that the Chora Runtime Environment contains a Garbage Collector that is responsible for disposing of all objects that are no longer in use. This Garbage Collector analyses the dependencies between all objects and recognizes which of these objects are no longer needed, in order to subsequently destroy them.

Garbage collection is automatically activated by the Chora Runtime Environment when needed. It is thus not possible to predict the exact point in time when unused objects will be destroyed. Depending on the target system and the implementation of the Garbage Collector, unused objects may remain in memory for some time until the Garbage Collector identifies and destroys them. As a result, possible calls to the destructor may be delayed accordingly. The point in time when the destructor is called can thus not be predicted.

Since destructors are optional, the absence of destructors does not lead to an error message; they are simply ignored. The Chora compiler recognizes whether or not a class has defined a destructor and generates the appropriate code to correctly deinitialize the objects of this class.

Destructors are used almost exclusively in driver classes that require direct access to the operating system and hardware. In this case, destructors take care of correctly deinitializing the hardware and maintaining access to the operating system during destruction of an object. Classes that do not require access to the hardware or operating system should not define destructors.


Copyright TARA Systems GmbH
www.embedded-wizard.de

Except for the above-mentioned restrictions in defining a destructor, there are no further differences between destructors and other methods. Thus, destructors may be inherited when deriving classes and may be overridden in the derived classes:

```
class SuperTimer : Core::Timer
{
  inherited method Done()
  {
    // Do some stuff here, but DON'T make a super call!!
    // super() !!!
  }
}
```

It must be noted that destructors are called only by the Chora Runtime Environment. This process begins with the object's most recently derived class and continues during deinitialization along the hierarchy of inheritance until the oldest base class is reached.

It can be inferred from the above example of the `SuperTimer` class that `SuperTimer` was derived from `Core::Timer`. If an object of class `SuperTimer` is destroyed at runtime, the `SuperTimer.Done()` method is executed first. Only after that is the inherited version of the method from the base class `Core::Timer` called. Please note that this process is automatically controlled by the Runtime Environment, so that use of the super call `super()` is superfluous and even undesirable.

Use of the optional keyword `native` allows destructors to be implemented in the form of native methods. Because destructors are intended primarily for direct access to the hardware and operating system, most destructors will therefore be native methods. See also "'native' statement" (chapter 5.17).

For more details see "Garbage collection" (chapter 11).

#### 4.1.5.7 ReInit re-constructors

The definition of a re-constructor is similar to an ordinary method. A re-constructor always has the name `ReInit`, expects no arguments and may not return a value:

```
class MenuItem
{
  method void ReInit()
  {
    Caption = GetCaptionForCurrentLanguage();
  }
}
```

The re-constructors are automatically invoked by the Chora Runtime Environment when the currently selected language changes to give the object the chance to request additional language dependent resources, etc. It is not necessary and, in fact, unadvisable to call the re-constructors directly from the Chora code! This can lead to inexplicable runtime errors:

```
var Menu::MenuItem item = new Menu::MenuItem;

// DON'T call re-constructors directly
timer.ReInit();
```

Since re-constructors are optional, the absence of re-constructors does not lead to an error message; they are simply ignored. The Chora compiler recognizes whether a class has defined a re-constructor or not, and generates the relevant re-initialization code, in order to re-initialize objects of this class correctly.

There are no differences between re-constructors and methods. Thus, re-constructors are inheritable in the derivation of classes and can be overridden in the derived classes:

```
class SuperMenuItem : Menu::MenuItem
{
  inherited method ReInit()
  {
    // Do some stuff here, but DON'T make super call!!!
    // super()
  }
}
```

It must merely be noted that the Runtime Environment automatically calls all of an object's re-constructors when an object is re-initialized. This process begins with the object's oldest base class and continues during re-initialization along the hierarchy of inheritance.

It can be inferred from the above example of the `SuperMenuItem` class that `SuperMenuItem` was derived from `Menu::MenuItem`. Thus, before the `ReInit()` method of the `SuperMenuItem` class is executed, the Runtime Environment takes care of re-initializing the `Menu::MenuItem` class and calling its `ReInit()` method. Only after that is the re-constructor of `SuperMenuItem` called. Please note that this process is automatically controlled by the Runtime Environment, so that the use of the super call `super()` is superfluous and even undesirable.

The use of the optional keyword `native` allows the re-constructors to be implemented in the form of native methods. See also "'native' statement" (chapter 5.17).

For more details see "Language selection" (chapter 9).

## 4.2 Constants

Constants are always introduced with the keyword `const`, followed by the data type, the name and value of the constant:

```
const string Caption = "Video Settings";
```

This defines a constant named `Caption`. The constant's data type has been given as `string`, whereby it is uniquely determined which values the constant may have. In this case, `string` means that the constant can store series of characters. After the assignment operator '=', there follows the actual value of the constant — the string `"Video Settings"`.

Once defined, constants may be used anywhere within the Chora program where constant expressions are allowed. One can, for example, use a constant to initialize variables. This ensures that the initial value of the variable is set in only one place, even if there are multiple variables scattered throughout the Chora program:

```
var string MyVariable1;
var string MyVariable2;

MyVariable1 = Sample::Caption;
MyVariable2 = Sample::Caption;
```

This example demonstrates the initialization of two variables with the value of the constant `Caption`, defined within the unit named `Sample`. Each time a constant is used in the Chora program, the name of the unit in which the constant has been defined must also be given. This ensures that the Chora compiler knows where to find the definition of the constants. Further, it prevents constant definitions with the same name in different units from causing name conflicts.

Following translation with the Chora compiler, the value of a constant may no longer be changed. An attempt to assign another value to a constant after the fact is not possible, and results in an error message from the Chora compiler:

```
Sample::Caption = "Sound Settings";
```

A constant always contains a value that fits the constant's data type. The possible data types are integrated into the Chora programming language. These data types allow the programmer to define constants with numbers, colors and even rectangles as values. For more details see "Data types" (chapter 6).

A constant may be language-dependent. This means that the constant can contain more than one value. Each of these values must be assigned to a particular language. The `Caption` constant shown above could, for example, be extended to include a second value for the `German` language:

```
const string Caption =
(
  German  = "Bildeinstellungen";
  Default = "Video Settings";
);
```

Please note that every language-dependent constant must contain a value for the `Default` language. Non-language-dependent constants are assigned a single value based on the `Default` language.

Please also note that the names of languages used in the definition must be defined in the project file. In this case, the project file must define the two languages `German` and `Default`.

The choice of the appropriate language-dependent value of a constant happens at the runtime of the Chora program and is completely transparent to the programmer. The programmer needs only to switch between the languages in order to set all constants to a value suitable to the language chosen. More details in "Language selection" (chapter 9).

Optionally, the `const` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
const string Caption = "Hello World!";
```

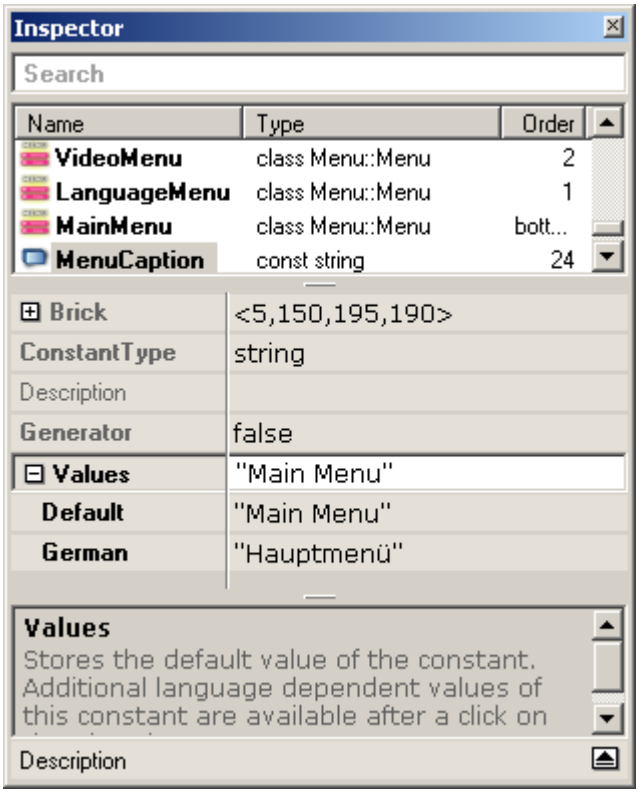

Copyright TARA Systems GmbH

For more details about the `$output` directive see "Control directives" (chapter 8.3).

In Chora it is possible to override a constant by so called constant variants. Constant variants provide a powerful technique for the extension and adaptation of constants. Each time the constant is used, Embedded Wizard determines automatically the appropriate variant and uses it instead → for more details see "Variants" (chapter 4.8).

Within the Embedded Wizard IDE the constants appear as bricks in the Unit Composer window:

The attributes of the constant can be set directly in the Inspector window, when the appropriate constant's brick has been previously selected:




## 4.3 Resources

Resources are always introduced with the keyword `resource` and the name of the resource itself:

```
resource Resources::Bitmap  Logo
{
  attr bitmapfile FileName = logo.png;
}
```

```
resource Resources::Font  Digits
{
  attr fontname    FontName  = Verdana;
  attr fontquality Quality   = Low;
  attr fontheight  Height    = 24;
  attr fontranges  Ranges    = '0'-'9';
}
```

Here, we have defined two resources, with the names `Logo` and `Digits`. The first resource contains a bitmap (resource class `Resource::Bitmap`). From the definition of this resource, the Chora compiler gets the name of a bitmap file, `logo.png`, reads this file, converts it automatically to a format suitable for the target system, and stores the result in the ANSI C code that is generated.

The second resource is approached in the same way. The Chora compiler recognizes from the resource class `Resources::Font` that it is dealing with a font here. In this case, the compiler attempts to read from the `Verdana` font all characters in the range '`0`' ... '`9`' and to convert these in a 24 pixel height font. So converted characters are stored in the generated ANSI C code.

The name of the resource class categorizes the resource as belonging to a certain class (bitmap, font, audio file, etc.) If a resource is needed at the runtime of a Chora program, the Resource Manager creates an object of this resource class automatically and loads it with the content of the resource. Through this object, the resource may be used now. It may, for example, be used for drawing operations, or one may query certain properties of the resource, for example the width or height of the bitmap:

```
var Resources::Bitmap bitmap;
var int32             width;
var int32             height;

// Get the resource object (the resource is loaded here)
bitmap = Sample::Logo;

// Query some attributes of the resource, etc.
width  = bitmap.Size.w;
height = bitmap.Size.h;
```

This example demonstrates how the resource named `Logo`, defined within the unit named `Sample`, is accessed. In accessing this resource, a `Resources::Bitmap` object is created and loaded with the content of the resource automatically. The object can then be accessed like any other Chora object.

To prevent the GUI program from loading the same resource several times, the Runtime Environment stores all resource objects that have already been loaded in a global table called 'resource map'. The management of this resource map is a part of the Resource Manager. The Resource Manager prevents one and the same resource from being loaded into memory more than once and thus ensures that resources that have already been loaded are shared within the Chora program.


Copyright TARA Systems GmbH
www.embedded-wizard.de

In the unit file, the definition of a resource is only a directive to the Chora compiler to read a certain external file (e.g. `logo.png` or a `Verdana` font) and to convert it into the format of the target system. The attributes necessary for this are introduced using the keyword `attr`. Depending on the type of resource, different attributes may be needed. If attributes are missing, or if they contain the wrong values, the Chora compiler reports an error message.

Each attribute consists of 3 elements: the meta-name, the name of the attribute and the value of the attribute. The meta-name and the name of the attribute are fixed for each resource class.

A resource can, as with constants, be language-dependent. This means that the resource may contain several versions. Each of these versions must be assigned to a particular language. One could, for example, extend the above-mentioned `Logo` resource by a second bitmap for the `German` language:

```
resource Resources::Bitmap Logo
{
  attr bitmapfile FileName =
  (
    German  = GermanBanner.png;
    Default = EnglishBanner.png;
  );
}
```

Please note that each language-dependent attribute of a resource must always contain a value for the `Default` language. Non-language-dependent resources are automatically assigned the sole value of the attribute for the language `Default`.

Please also note that the names of languages must be defined in the project file. In this case, the project file must define the two languages `German` and `Default`.

The choice of the suitable language-dependent resource occurs automatically at the runtime of the Chora program and is completely transparent to the programmer. The programmer needs only to switch between the languages in order to set all resources to the language selected. More details in "Language selection" (chapter 9).

During translation a resource, the Chora compiler makes use of special resource converters. Each resource class requires its own resource converter, and each Platform Package must contain its own set of such converters. This technique guarantees that the definitions of a resource may be ideally transferred to the format of the target platform and that the Chora compiler may be improved in the future to include new resources (e.g. sound, animated GIFs, etc.). The current version of the Chora compiler contains only a bitmap and a font resource converter.

Optionally, the `resource` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
resource Resources::Bitmap Logo
{
  ...
}
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).


Copyright TARA Systems GmbH

In Chora it is possible to override a resource by so called resource variants. Resource variants provide a powerful technique for the extension and adaptation of resources. Each time a resource is used, Embedded Wizard determines automatically the appropriate variant and uses it instead → for more details see "Variants" (chapter 4.8).

### 4.3.1 Attributes of a bitmap resource

A resource of the class `Resources::Bitmap` supports the following attributes:

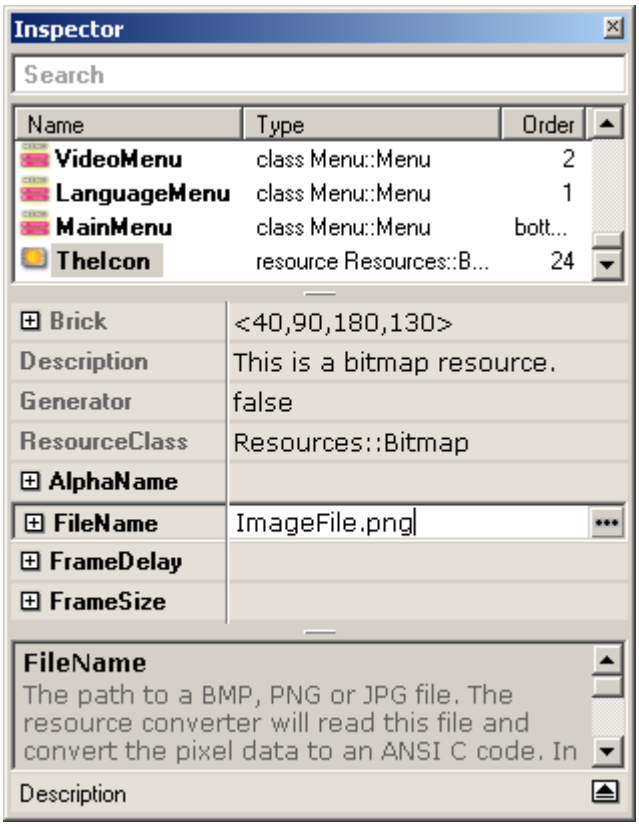
| Attribute name | Description |
|---|---|
| `FileName` | This attribute contains the name of a bitmap file containing the bitmap color information. The following image file formats are supported: `*.BMP`, `*.PNG`, `*.GIF` and `*.JPG`. The resource converter will read this file and convert the bitmap pixel data to an ANSI C code.<br><br>In case of a `.PNG` file (or `.GIF`), the resource converter also evaluates the transparency information stored in the file's alpha channel.<br><br>If no alpha information is available in the bitmap file, you can provide it in a separate bitmap file specified in the following `AlphaName` attribute.<br><br>If `FileName` attribute was ommited and an `AlphaName` attribute specified, the resulting bitmap resource is considered as alpha bitmap only → the bitmap will contain transparance information only. Please note, this feature is supported together with the new Graphics Engine 2.0 only.<br><br>If the file name contains a relative path, the bitmap file will be looked for relative to the directory of the unit file, where the bitmap resource is defined. |
| `AlphaName` | This optional attribute contains the name of an additional `*.BMP`, `*.PNG`, `*.GIF` or `*.JPG` file that is interpreted as the transparence mask (alpha channel) for the origin bitmap specified in the attribute `FileName`.<br><br>If `FileName` attribute was ommited, the resulting bitmap resource is considered as alpha bitmap only → the bitmap will contain transparance information only.<br><br>If this optional bitmap is specified, the resource converter evaluates the gray values of its pixel as transparence information for the corresponding pixel of the origin bitmap. In this case the black color stands for a full transparent pixel and the white color corresponds to a pixel with a full opacity. Please note, this feature is supported together with the new Graphics Engine 2.0 only.<br><br>If the file name contains a relative path, the bitmap file will be |

| | |
|---|---|
| | looked for relative to the directory of the unit file, where the bitmap resource is defined. |
| `FrameSize` | This optional attribute is useful in case of a bitmap, which consists of several small images (frames). Multi-frame bitmaps can serve as image-lists or they can contain short animation sequences.<br><br>Within the origin bitmap, the frames are arranged in tiles, side by side. This attribute specifies the size in pixel of a single frame. If you design multi-frame bitmaps, you can arrange the frames within the bitmap in rows and/or in columns.<br><br>If this attribute is omitted, the entire content of the bitmap is assumed as the only one frame. In this case `FrameSize` is set automatically to the size of the origin bitmap. |
| `FrameDelay` | This optional attribute is useful in case of a bitmap, which contains an animation sequence. To achieve this, ensure first, that the valid size of a single animation frame is specified in the `FrameSize` attribute.<br><br>`FrameDelay` specifies the delay in milliseconds between two animation frames. In this manner, the animation speed and the sequence of animation frames are stored together within a single bitmap resource.<br><br>If this attribute is omitted, `FrameDelay` is set to `0`, so the content of the bitmap is signed as 'not animated'. |

**Table 4–1**

Within the Embedded Wizard IDE the bitmap resources appear as bricks in the Unit Composer window:

The attributes of the resource can be set directly in the Inspector window, when the appropriate resource's brick has been previously selected:




## 4.3.2 Attributes of a font resource

A resource of the class `Resources::Font` supports the following attributes:

| Attribute name | Description |
|---|---|
| `FontName` | This attribute contains the name of a Microsoft Windows font. The desired font must exist and already be installed under Windows. Only TrueType fonts are allowed. Any attempt to convert a bitmap font results in an error. |
| `Quality` | The `Quality` attribute describes the desired quality of the converted font. The programmer can choose between `Low`, `Medium` and `High`. High quality improves the readability of the font through anti-aliasing, but raises the memory requirement of the font resource.<br>Quality is only a suggestion. Whether the font is indeed anti-aliased is determined by the resource converter. |
| `Height` | The `Height` attribute defines the desired height of the font in pixels. |
| `Bold` | The `Bold` attribute controls the weight of the font. If this attribute is set `true`, a bold version of the font is converted. |
| `Italic` | The `Italic` attribute determines, whether an italic version of the font should be converted. The attribute expects a `true` or a |

| | |
|---|---|
| | `false` value. |
| `AspectRatio` | The `AspectRatio` attribute defines the desired aspect ratio of the font in the range from 0.25 to 4.0. The default value is 1.0 - in this case the aspect ratio of the converted font corresponds to the origin design of the font. |
| `Ranges` | With the attribute `Ranges` a set of characters can be specified for the font conversion. Only the characters listed in this attribute are affected.<br><br>The simplest set consists of a single character (e.g. `'0'`). To define a range of characters the first and the last character has to be specified (e.g. `'0'-'9'`).<br><br>Instead of the character literals it is also possible to use the `\x` hexadecimal notation (e.g. `'\x0030'-'\x0039'`) or decimal notation (e.g. `48-58`).<br><br>The `Ranges` attribute allows the programmer to specify more than one character range for a single font resource. In this case the ranges are separated by the comma sign (e.g. `'A'-'Z','a'-'z','0'-'9','\x0020'`). All these characters will be then affected by the font conversion. In this manner different character ranges for different languages can be included into a single font resource.<br><br>Additionally, it is possible to map a single range of characters to another character code. For this purpose the desired destination character is specified after the colon sign (e.g. `'a'-'z':'A'` forces the font converter to convert the lower case characters and to map them to the range of the upper case characters). This mapping is optionally and it can be applied to each range separately.<br><br>Alternatively the attribute can specify a path referring to a text file containing the desired font character ranges (e.g. `file.txt`). The syntax of this file should correspond to the syntax described above. If the file name contains a relative path, the file will be looked for relative to the directory of the unit file, where the font resource is defined. |

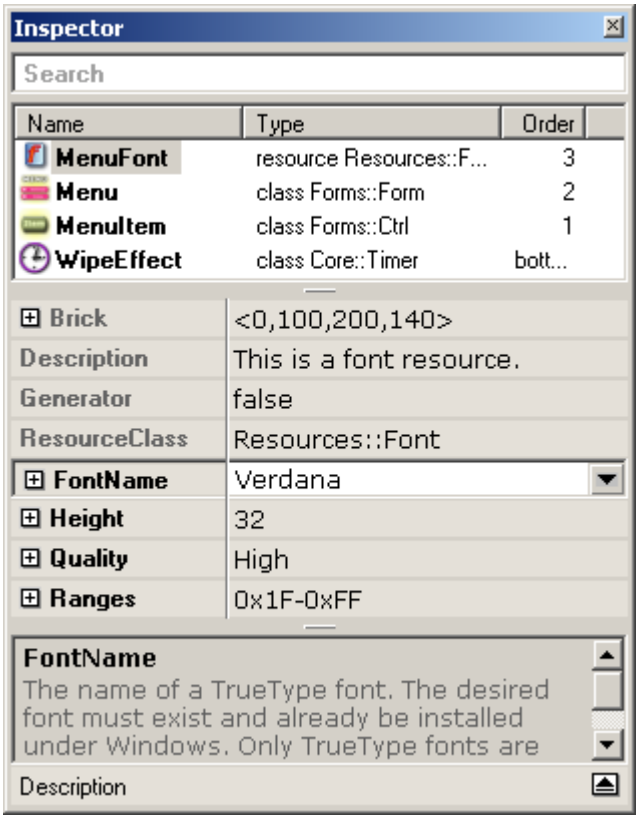
**Table 4–2**

Note: The converted font resource includes additionally the default character. This character will be displayed on the screen instead of the characters, which are not contained in the font resource. The code of the default character is specified in the TrueType font. The font converter takes over this default character automatically; it is not necessary to specify it explicitly in the `Ranges` attribute.

Additionally please note, that Embedded Wizard handles multilingual font resources in a different way, as it is habitual to do with e.g. bitmap resources. Instead of selecting an appropriate language variant of a font resource, Embedded Wizard combines the several language variants together and considers them as a single large 'virtual' font. In this manner the programmer can create complex fonts, which can be used to display text consisting of absolutely different character sets all at once. For example, the programmer can create a font resource, which combines the Latin character set from the Arial 18 font, the Greek character set from the Verdana 16 font and some Japanese signs from the Arial Unicode 16 font. This font resource can then be used to display text for all the supported languages – independently of the currently selected language.

Within the Embedded Wizard IDE the font resources appear as bricks in the Unit Composer window:

The attributes of the resource can be set directly in the Inspector window, when the appropriate resource's brick has been previously selected:




## 4.4 Auto objects

Auto objects are introduced by the keyword `autobject`, followed by the objects class, the name and an optional a set of initialization values (presets) of the object. Auto objects can only be defined in scope of a unit:

```
autobject System::AudioController Audio
{
  preset Volume = 50;
  preset Stereo = true;
}
  ...
```

In this example, the definition of the unit contains an auto object named `Audio`. The auto object is an instance of the class `System::AudioController`. The object's two properties, `Volume` and `Stereo`, should be initialized with the value `50` and `true`.

As soon as the auto object is accessed the first time, it is instantiated. The `preset` keywords force the properties of the object to be re-initialized. If a property is not explicitly re-initialized using `preset`, the property retains the value originally defined in the object's class. Thus only properties that have been re-initialized using `preset`, are given new values.

When the properties are re-initialized, the corresponding onset methods are automatically invoked, so that the object can react to the change in properties. Note that the name of the re-initialized property must fit to one of the properties defined in the object's class. If this is not the case the Chora compiler reports an error message.

An auto object does not necessarily have to contain re-initialization values for its properties. Often it suffices to use the property values defined in the object's class:

```
autobject System::AudioController Audio;
```

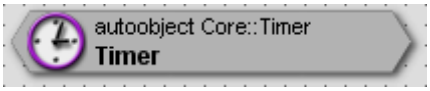
In this case, all properties of the object retain their original value. No properties are re-initialized.

The auto object can be accessed by its name in scope of the superior unit. By using this full name, the properties of the object can be read or modified and methods can be invoked:

```
TheUnit::Audio.Volume = 100;
TheUnit::Audio.DoSomething();
```

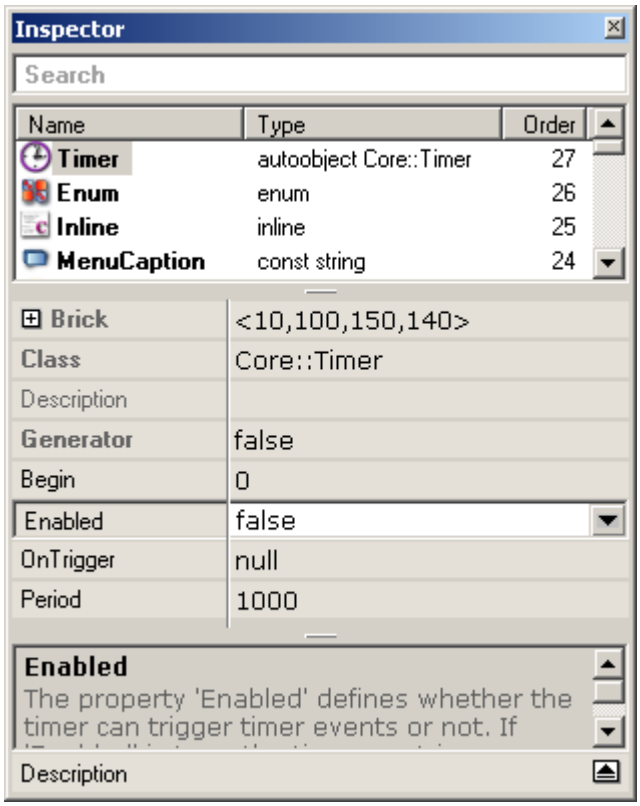
When an auto object is not in use anymore, the Garbage Collector automatically disposes it. The following access to the disposed object causes it to be instantiated and initialized again. This is the major difference between auto objects and ordinary global objects, which are known from other programming languages.

Within the Embedded Wizard IDE auto objects always appear as bricks in the Unit Composer window:




The attributes of the object can be set directly in the Inspector window, when the appropriate object's brick has been previously selected:


Copyright TARA Systems GmbH
www.embedded-wizard.de




## 4.5 Inline code

An inline definition must always be introduced using the keyword `inline`, followed by the name of the definition and a user-defined code block:

```
inline Includes
{
  #include <rtos_api.h>
  #include <stdio.h>
}
```

Here, a code block is defined containing two ANSI C `#include` directives. This code block is automatically inserted into the generated ANSI C file without any changes.

The use of inline definitions allows a mixture of Chora and native code. However, it prevents platform-independent GUI development. In practice, inline definitions should be used only in developing driver classes. A driver class serves as an interface between the GUI and the underlying target system. By adapting the driver class, the GUI program can be ported to other target platforms.

In order to differentiate between different target platforms within an inline definition, it is necessary to use the Chora preprocessor directives:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
inline DebuggerOutput
{
  $if ( $platform == Tara.Win32.* )
    static void Debug( char* aMsg )
    {
      printf( aMsg );
    }
  $else
    static void Debug( char* aMsg )
    {
      __error_msg( aMsg,0 );
    }
  $endif
}
```

By using the `$if`, `$else`, `$elseif` and `$endif` Chora preprocessor directives, part of the code block, depending on the target platform, can be ignored or taken over. If the condition in an `$if` or `$elseif` directive is fulfilled (is true), the following part of the code block is taken over. If the condition is not fulfilled, the code block is ignored. Details of the Chora preprocessor are found in "Chora preprocessor" (chapter 8)

The inline code blocks are placed in the generated ANSI C file according to their sequence (Z order) within the unit file, together with other definitions, such as classes, constants, resources, and so on.

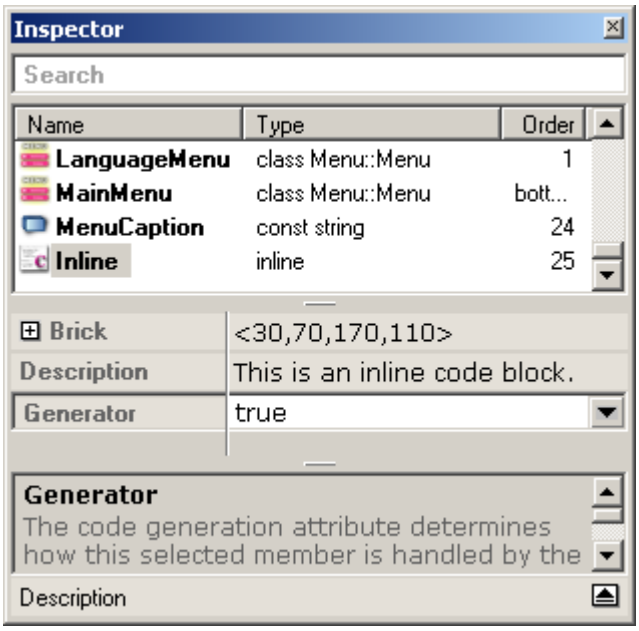
Optionally, the `inline` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
inline DebuggerOutput
{
  ...
}
```

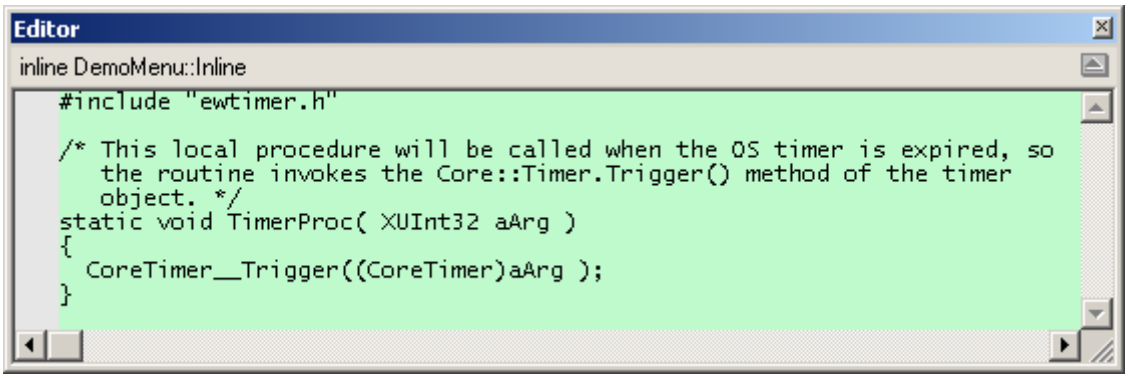
For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE the inline code blocks appear as bricks in the Unit Composer window:

The attributes of the inline code block can be set directly in the Inspector window, when the appropriate brick has been previously selected:




The content of the inline code block can be implemented in the Editor window, after the block's brick has been double clicked:


Editor
inline DemoMenu::Inline
#include "ewtimer.h"
/* This local procedure will be called when the OS timer is expired, so
the routine invokes the Core::Timer.Trigger() method of the timer
object. */
static void TimerProc( XUInt32 aArg )
{
CoreTimer__Trigger((CoreTimer)aArg );
}

## 4.6 Enumerations

Enumerations are always introduced using the keyword `enum`, followed by the name of the enumeration itself:

```
enum AlignH
{
  item Left;
  item Center;
  item Right;
}
```

Here we have defined a new enumeration named `AlignH`. The enumeration consists of 3 elements, so-called enumerators, with the names `Left`, `Center` and `Right`. An enumeration is a user-defined data type. This data type may be used to declare variables that may have a specific value given by the enumerator list:

```
var Graphics::AlignH align;
align = Graphics::AlignH.Center;
```

The above example demonstrates how a variable of the user-defined enumeration data type `Graphics::AlignH` is declared. After the declaration, the variable may contain one and only one element of this enumeration. Other values are not allowed and are seen as errors by the Chora compiler — as, for example, this attempt to combine two different enumerations in a single assignment:

```
var Graphics::AlignH align;
align = Sample::OtherOptions.DVD;
```

An enumeration data type is very practical if a variable may be given only one value from a limited list. Any other values that are not listed may not be stored in the variable.

During the code generation, Embedded Wizard converts automatically all affected `enum` definitions into code artifacts appropriate for the respective target system. While this task, the Code Generator assigns numeric values to the individual items of the generated `enum` definitions. Usually this process starts with the value 0 for the first item, 1 for the second, 2 for the third and so far. If desired, the developer can control this automatic enumeration. In this case it is possible to map Chora `enum` definitions to some corresponding definitions already existing in the target system.

In order to change the automatic enumeration, it is necessary to assign a designated numeric value to the appropriate item. In such case the Code Generator takes over the given number and uses it for the enumeration of all following items:

```
enum AlignH
{
  item Left;
  item Center = 10;
  item Right;
}
```

The above example demonstrates the explicitly designation of a numeric value to an item within an `enum` definition. In this case the second item `Center` has become a value `10`. This will force the Code Generator to use the specified value for all occurrences of the item within the generated code. As mentioned above the Code Generator also uses the specified value for the enumeration of all following items. Therefore the third item `Right` will receive the value `11`.

Optionally, the `enum` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

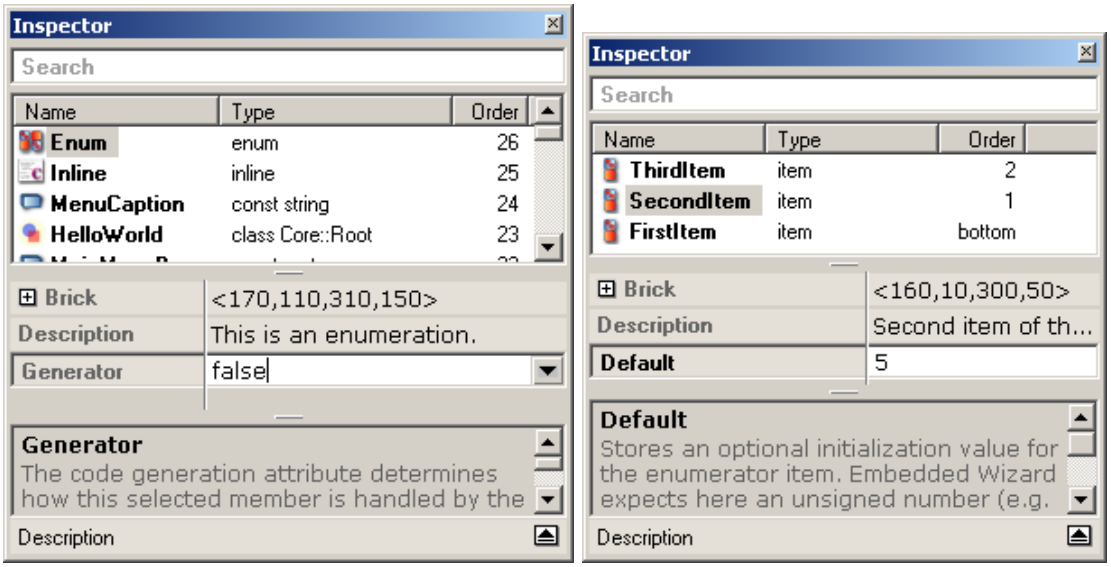
```
$output true
enum AlignH
{
  ...
}
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE the `enum` definitions and their item enumerator members appear as bricks in the Composer window:


Copyright TARA Systems GmbH
www.embedded-wizard.de

The attributes of these definitions can be set directly in the Inspector window, when the appropriate brick has been previously selected:




## 4.7 Sets

Sets are always introduced using the keyword `set`, followed by the name of the set itself:

```
set States
{
  item Visible;
  item Selected;
  item Highlighted;
}
```

Here, we have defined a new set named `States`. The set consists of 3 elements, the so-called enumerators with the names `Visible`, `Selected` and `Highlighted`. A set is a user-defined data type. This data type can be used to declare variables that will contain any combination of enumerators from the set:

```
var Sample::States states;
states = Sample::States[ Visible, Highlighted ];
```

The above example demonstrates how a variable with the user-defined set data type `Sample::States` is declared. After the declaration, the variable may contain only elements from this set. Other values are not allowed and will be seen as errors by the Chora compiler — as, for example, in this attempt to combine two different sets in a single assignment:

```
var Sample::States states;
states = Sample::OtherStates[ Enabled, TopMost ];
```

The variable may also contain an empty set. In this case, no elements of the set are listed; the brackets are empty:

```
states = Sample::States[];
```

A set data type is very practical if a variable may be given a combination of particular values from a limited list. No other values not contained in the list may be stored in the variable. Special operators allow the programmer to perform operations on sets. He can determine the intersection or combination of two sets, for example. Further details about operators may be found in "Operators" (chapter 7).

During the code generation, Embedded Wizard converts automatically all affected `set` definitions into code artifacts appropriate for the respective target system. While this task, the Code Generator assigns numeric values to the individual items of the generated `set` definitions. Usually this process starts with the value 1 for the first item, 2 for the second, 4 for the third, 8 for the fourth and so far – up to 32 items. If desired, the developer can control this automatic enumeration. In this case it is possible to map Chora `set` definitions to some corresponding definitions already existing in the target system.

In order to change the automatic enumeration, it is necessary to assign a designated numeric value to the appropriate item. In such case the Code Generator takes over the given number and uses it for the enumeration of all following items:

```
set States
{
  item Visible;
  item Selected = 0x100;
  item Highlighted;
}
```

The above example demonstrates the explicitly designation of a numeric value to an item within an `set` definition. In this case the second item `Selected` has become a value `0x100 (256)`. This will force the Code Generator to use the specified value for all occurrences of the item within the generated code. As mentioned above the Code Generator also uses the specified value for the enumeration of all following items. Therefore the third item `Highlighted` will receive the value `0x200 (512)`.

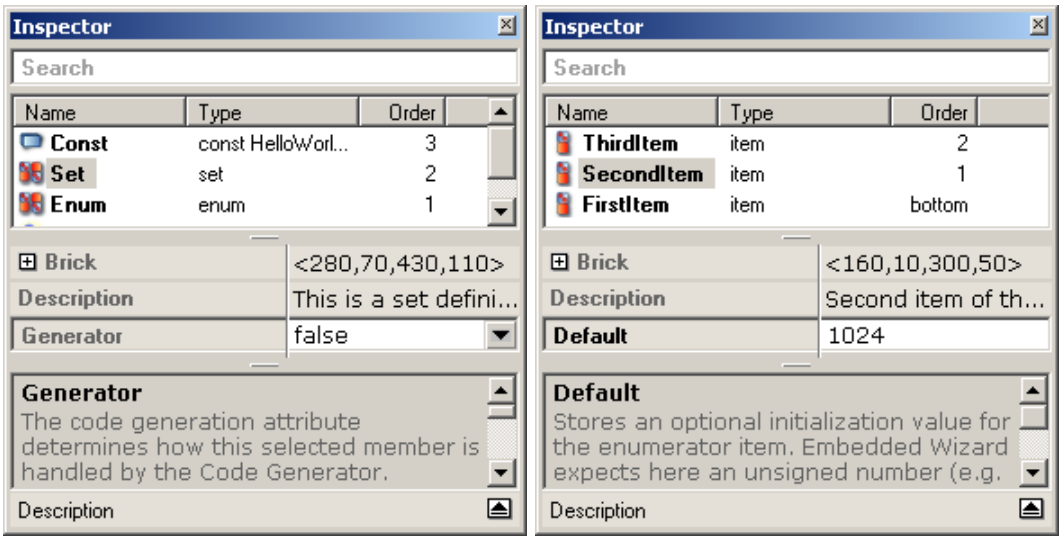
Optionally, the `set` definition may be prefixed with the `$output` directive. This directive controls the code generation for the affected definition; it is generated if the condition of the directive is fulfilled:

```
$output true
set States
{
  ...
}
```

For more details about the `$output` directive see "Control directives" (chapter 8.3).

Within the Embedded Wizard IDE the `set` definitions and their item enumerator members appear as bricks in the Composer window:

The attributes of these definitions can be set directly in the Inspector window, when the appropriate brick has been previously selected:




## 4.8 Variants

The programming language Chora provides a unique feature called 'variants'. This feature allows the development of GUI applications, with multiple different appearances and behaviors. The selection of the desired appearance and behavior is done during the code generation or at the runtime. Depending on them, we differentiate between static and dynamic variants.

The idea behind the variants is based on the object-oriented concept of derivation. This is realized in a way similar to the class inheritance – the affected parts of the GUI can be simple derived and adapted. The one big difference is, that variants do override (substitue) the origin definitions they are derived from. If at the runtime the definition is used, Embedded Wizard does handle with the variant instead.

Following definitions can be substituted by derived variants:

- Classes → In the derived class variant, the inherited members can be adapted, new members can be added and existing methods can be overridden in order to change the behavior of the class → see "Class Variants" (chapter 4.8.1).
- Constants → In the derived constant variant, the inherited language dependent values can be modified and new language variants can be added to the constant → see "Constant Variants" (chapter 4.8.2).
- Resources → Similarly to the constant variants, the language dependent values of a resource can also be modified or extended → see "Resource Variants" (chapter 4.8.3).
- Auto objects → In derived object variant, the inherited properties can obtain modified initialization values → see "Auto object Variants" (chapter 4.8.4).

By using variants, the origin (substituted) definition itself does always remain unchanged and all desired modifications are implemented in the derived variants only. This simplifies the customization process significantly. The customer can now adapt the GUI application to his requirements without the necessity of any modifications on the software delivered by his supplier.


Copyright TARA Systems GmbH
www.embedded-wizard.de

Unlike the ordinary class inheritance, each variant is signed with a variant condition. This condition determines when the variant will be used in order to substitute its origin definition. Only if the condition is fulfilled, the affected variant is applied. In this manner, multiple variants can be implemented and the user can switch between them. The condition accepts a list of profile and style names. The condition is fulfilled, if one of the specified profiles or styles is selected.

The variant condition is evaluated as well as during the code generation and at the runtime. If the condition is fulfilled already during the code generation, the affected variant does substitute its origin definition permanently – it can't be switched at the runtime anymore. Such kind of a variant is called 'static' variant. Usually its condition depends on a profile, which is then selected for the code generation.

The second kind is called 'dynamic' variants. Dynamic variants can be switched at the runtime, if the appropriate variant condition depends on one of the active styles. The activation and deactivation of styles is performed by assigning a set of style names to the global built-in Chora variable `styles`. For example, the both styles `AquaLook` and `WideScreen` are activated in the following manner:

```
styles = [AquaLook, WideScreen];
```

Alternatively the variant condition can be set to the literal `true`. In this case the variant substitutes its origin definition permanently, independent from any profiles or styles. For more details about profiles and styles → see "Profiles and macros" (chapter 3.2) and "Styles" (chapter 3.4)

If the variant condition is set to the literal `default`, the variant substitutes its origin definition only when no other variant is available.

Alternatively the variant condition can be set to the literal `true`. In this case the variant substitutes its origin defintion permanently, independent

If the condition of a variant is never fulfilled, no code is generated for the affected variant. This automatic elimination reduces the memory footprint of Chora programs.

If two variants of the same origin definition depend on the same variant condition, a Chora compiler error is reported, because the Embedded Wizard is not able to distinguish between those both variants.

It is also possible to derived variants from other variants. So complex variant hierarchy is possible. For more details → see "Usage of Variants" (chapter 10).

## 4.8.1 Class Variants

Class variants are always introduced by the keyword `vclass`, followed by the name of the variant and the name of the origin class (the ancestor):

```
vclass AquaItem : Menu::MenuItem
{
  inherited property color Color  = #0088FFFF;

  inherited method HandleEvent()
  {
    // Handle events for the item ...
  }
}
```

Using this definition, we have created a variant `AquaItem` of a class `Menu::MenuItem`. In the variant, the value of the `Color` property and the logic of the `HandleEvent()` method have been modified. These modifications do affect the appearance and the behaviour of the origin class `Menu::MenuItem`.

If at the runtime an object of the origin class `Menu::MenuItem` is created, Embedded Wizard will automatically use `AquaItem` class variant instead. The developer does not need to take care about it:

```
var Menu::MenuItem theObject = new Menu::MenuItem;
```

The Embedded Wizard performs the selection of the class variant automatically. An explicit usage of a class variant is not allowed. The following example will cause a Chora compiler error because `MyMenu::AquaItem` is a class variant - to create an object a 'real' class is expected:

```
var MyMenu::AquaItem theObject = new MyMenu::AquaItem;
```

The approach of the implementation of a class variant does not differ from the one of a 'real' class. Derived members can be overridden and new members can be added in the same way as this is described in "Classes" (chapter 4.1).

Optionally, the `vclass` definition may be prefixed with the `$variant` directive. This directive determines the condition when the variant is used in order to substitute its origin class; the variant is used only if the condition of the directive is fulfilled. For example the following variant is used if the `Win32` profile is selected for the code generation. In all other cases, the variant is ignored:

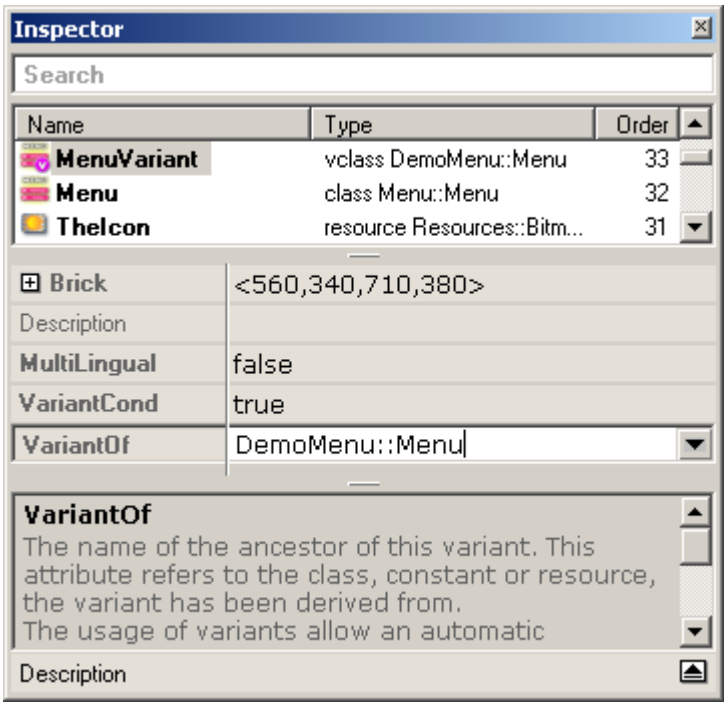
```
$variant Win32
vclass AquaItem : Menu::MenuItem
{
  ...
}
```

The `$variant` directive allows the implementation of multivariant classes. A single class can be overridden by multiple variants. Which variant is finally used, is determined during the code generation and at the runtime. In this case we distinguish between static and dynamic variants. Static variants depend on profiles and dynamic variants on styles. For more details about the `$variant` directive see "Control directives" (chapter 8.3).

In Chora it is also possible to derive a class variant from another class variant. This results in very powerful instrument for the development of multivariant GUI applications.

Within the Embedded Wizard IDE class variants appear as class bricks extended by a small icon in the Unit Composer window:

A double click on the class variant brick will show its class members and if the class contains graphical objects, these objects are also drawn on the screen. The attributes of the class variant can be set directly in the Inspector window, when the appropriate variant brick has been previously selected:




## 4.8.2 Constant Variants

Constant variants are always introduced by the keyword `vconst`, followed by the name of the variant and the name of the origin constant (its ancestor):

```
vconst MyCaption : Menu::Caption = "Video Settings";
```

Using this definition, we have created a variant `MyCaption` of a constant `Menu::Caption`. The variant inherits all language dependent values from the origin constant and overrides the default value with the string `"Video Settings"`. Please note, the constant variant does not specify the data type of its content. This data type is always inherited from the origin constant. Within a constant variant only the values of the constant can be changed.

If at the runtime the constant `Menu::Caption` is used, Embedded Wizard will automatically use the `MyCaption` constant variant instead. The developer does not need to take care about it:

```
Dialog.Caption = Menu::Caption;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

The Embedded Wizard performs the selection of the constant variant automatically. An explicit usage of a constant variant is not allowed. The following example will cause a Chora compiler error because `MyMenu::MyCaption` is a variant of the constant and not a 'real' constant:

```
Dialog.Caption = MyMenu::MyCaption;
```

Like a 'real' constant, a constant variant may store multiple, language dependent values. Similarly to the class inheritance, a constant variant inherits all values from its ancestor. If necessary the inherited values can be modified or new values can be added:

```
const string Caption =
(
  Default = "Hello!";
  German  = "Guten Tag!";
);

vconst MyCaption : Menu::Caption =
(
  Default = "Good Morning!";
  French  = "Bonjour!";
);
```

In the example above, the variant `MyCaption` extends the origin constant by a new value for the language `French` and overrides the inherited default value from `"Hello!"` to `"Good Morning!"`. The value for the `German` language still keeps unchanged. From the programmer's point of view, the constant `Caption` contains now 3 language dependent values. Please note, the language `French` is new added to the constant. In this way, the customer can customize existing constants and adapt them to new languages.

The choice of the appropriate language-dependent value of a constant happens at the runtime of the Chora program and is completely transparent to the programmer. The programmer needs only to switch between the languages in order to set all constants to a value suitable to the language chosen. More details in "Language selection" (chapter 9). The concepts of language-dependent and multivariant constants do complement one another → see "Usage of Variants" (chapter 10).

Beside the few differences described above, the approach of the definition of a constant variant does not differ from the one of a 'real' constant as this is described in "Constants" (chapter 4.2).

Optionally, the `vconst` definition may be prefixed with the `$variant` directive. This directive determines the condition when the variant is used in order to substitute its origin constant; the variant is used only if the condition of the directive is fulfilled. For example the following variant is used if the `Win32` profile is selected for the code generation. In all other cases, the variant is ignored:

```
$variant Win32
vconst MyCaption : Menu::Caption =
(
  ...
);
```

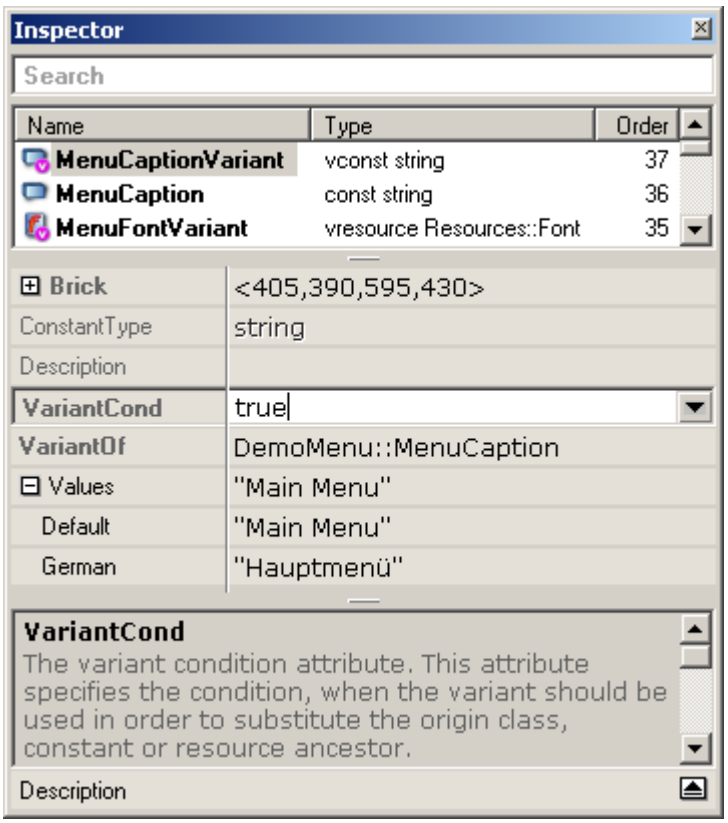

Copyright TARA Systems GmbH
www.embedded-wizard.de

The `$variant` directive allows the definition of multivariant constants. A single constant can be overridden by multiple variants. Which variant is finally used, is determined during the code generation and at the runtime. In this case we distinguish between static and dynamic variants. Static variants depend on profiles and dynamic variants on styles. For more details about the `$variant` directive see "Control directives" (chapter 8.3).

In Chora it is also possible to derive a constant variant from another constant variant. This results in a very powerful instrument for the development of multivariant GUI applications and it simplifies the customization of existing software.

Within the Embedded Wizard IDE constant variants appear as constant bricks extended by a small icon in the Unit Composer window:

The attributes of the variant can be set directly in the Inspector window, when the appropriate variants brick has been previously selected:




## 4.8.3 Resource Variants

Resource variants are always introduced by the keyword `vresource`, followed by the name of the variant and the name of the origin resource (its ancestor):

```
vresource MyLogo : Menu::Logo
{
  attr FileName = MyLogo.png;
}
```

Using this definition, we have created a variant `MyLogo` of a resource `Menu::Logo`. The variant inherites all resource attributes and their language dependent values from the origin resource and overrides the default value of the attribute `FileName` with `MyLogo.png`. Please note, the resource variant does not specify the resource class nor the meta-name of the overridden attributes. This information is always inherited from the origin resource. Within a resource variant only the values of its attributes can be changed.

If at the runtime the resource `Menu::Logo` is used, Embedded Wizard will automatically use the `MyLogo` resource variant instead. The developer does not need to take care about it:

```
Icon.Bitmap = Menu::Logo;
```

The Embedded Wizard performs the selection of the resource variant automatically. An explicit usage of a resource variant is not allowed. The following example will cause a Chora compiler error because `MyMenu::MyLogo` is a variant of the resource and not a 'real' resource:

```
Icon.Bitmap = MyMenu::MyLogo;
```

Like a 'real' resource, a resource variant may store multiple, language dependent values for every resource attribute. Similarly to the class inheritance, a resource variant inherits all values from its ancestor. If necessary the inherited values can be modified or new values can be added:

```
resource Resources::Bitmap Logo
{
  attr bitmapfile FileName =
  (
    Default = EnglishLogo.png;
    German  = GermanLogo.png;
  );
}

vresource MyLogo : Menu::Logo =
{
  attr FileName =
  (
    Default = MyOwnLogo.png;
    French  = FrenchLogo.png;
  );
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

In the example above, the variant `MyLogo` extends the attribute `FileName` of the origin resource by a new value for the language `French` and overrides the inherited default value from `EnglishLogo.png` to `MyOwnLogo.png`. The value for the `German` language still keeps unchanged. From the programmer's point of view, the resource `Logo` contains now 3 languages. Please note, the language `French` is new added to the resource. In this way, the customer can customize existing resources and adapt them to new languages.

The choice of the suitable language-dependent resource occurs automatically at the runtime of the Chora program and is completely transparent to the programmer. The programmer needs only to switch between the languages in order to set all resources to the language selected. More details in "Language selection" (chapter 9). The concepts of language-dependent and multivariant resources do complement one another → see "Usage of Variants" (chapter 10).

Beside the few differences described above, the approach of the definition of a resource variant does not differ from the one of a 'real' resource as it is described in "Resources" (chapter 4.3).

Optionally, the `vresource` definition may be prefixed with the `$variant` directive. This directive determines the condition when the variant is used in order to substitute its origin resource; the variant is used only if the condition of the directive is fulfilled. For example the following variant is used if the `Win32` profile is selected for the code generation. In all other cases, the variant is ignored:

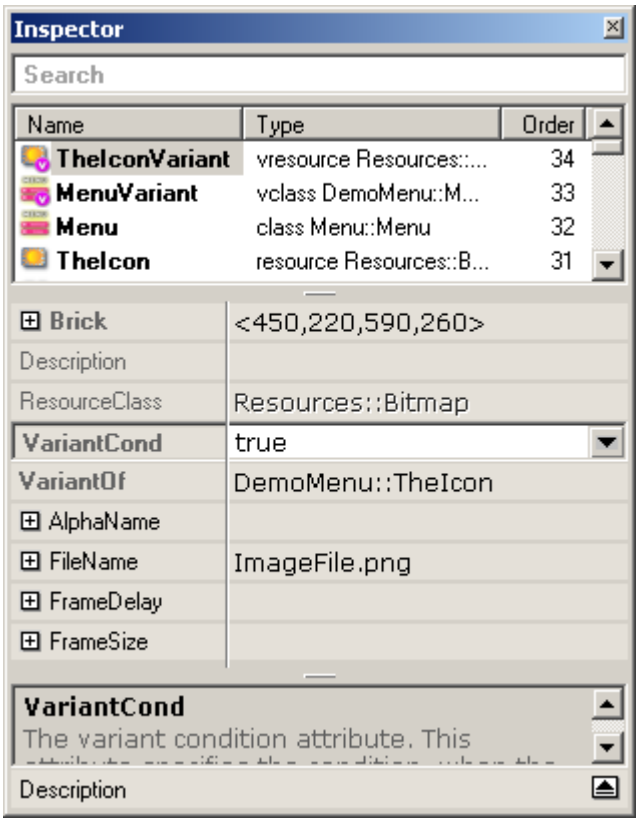
```
$variant Win32
vresource MyLogo : Menu::Logo
{
  ...
};
```

The `$variant` directive allows the definition of multivariant resources. A single resource can be overridden by multiple variants. Which variant is finally used, is determined during the code generation and at the runtime. In this case we distinguish between static and dynamic variants. Static variants depend on profiles and dynamic variants on styles. For more details about the `$variant` directive see "Control directives" (chapter 8.3).

In Chora it is also possible to derive a resource variant from another resource variant. This results in very powerful instrument for the development of multivariant GUI applications and it simplifies the customization of existing software.

Within the Embedded Wizard IDE resource variants appear as resource bricks extended by a small icon in the Unit Composer window:

The attributes of the variant can be set directly in the Inspector window, when the appropriate variants brick has been previously selected:




## 4.8.4 Auto object Variants

Auto object variants are always introduced by the keyword `vautoobject`, followed by the name of the variant and the name of the origin auto object (the ancestor):

```
vautoobject AudioLoud : TheUnit::Audio
{
  preset Volume = 100;
}
```

Using this definition, we have created a variant `AudioLoud` of an auto object `TheUnit::Audio`. In the variant, the initialization value of the `Volume` property has been modified. This modification do affect the behavior of the origin auto object `TheUnit::Audio`.

If at the runtime the origin auto object `TheUnit::Audio` is accessed, Embedded Wizard will automatically use `AudioLoud` object variant instead. The developer does not need to take care about it:

```
var int32 theVolume = TheUnit::Audio.Volume;
```

The Embedded Wizard performs the selection of the auto object variant automatically. An explicit usage of an auto object variant is not allowed. The following example will cause a Chora compiler error because `TheUnit::AudioLoud` is an auto object variant – instead of this the 'real' auto object would be expected:

```
var int32 theVolume = TheUnit::AudioLoud.Volume;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Optionally, the `vautoobject` definition may be prefixed with the `$variant` directive. This directive determines the condition when the variant is used in order to substitute its origin auto object; the variant is used only if the condition of the directive is fulfilled. For example the following variant is used if the `Win32` profile is selected for the code generation. In all other cases, the variant is ignored:

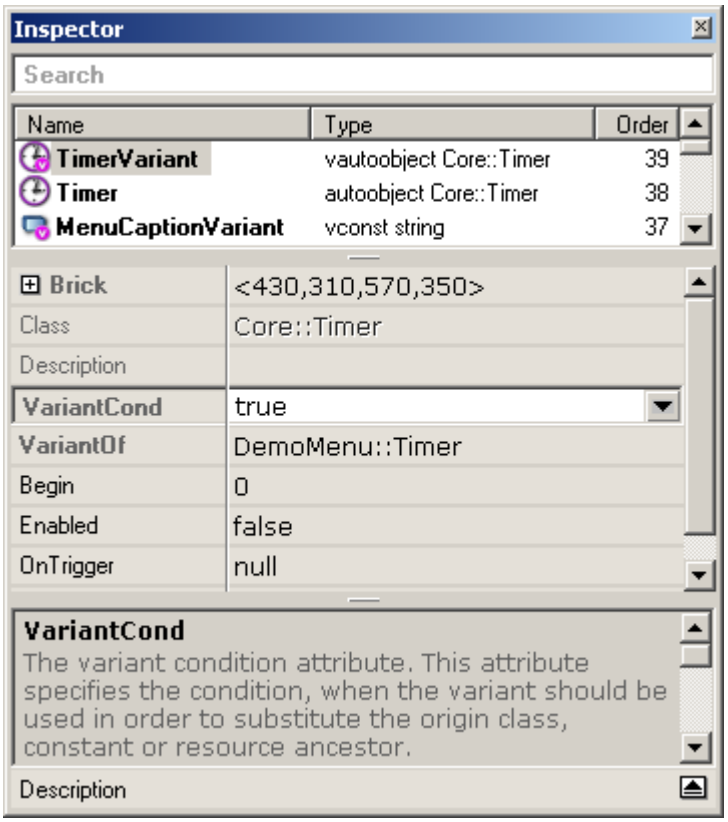
```
$variant Win32
vautoobject AudioLoud : TheUnit::Audio
{
  ...
}
```

The `$variant` directive allows the implementation of multivariant auto obejcts. A single auto object can be overridden by multiple variants. Which variant is finally used, is determined during the code generation and at the runtime. In this case we distinguish between static and dynamic variants. Static variants depend on profiles and dynamic variants on styles. For more details about the `$variant` directive see "Control directives" (chapter 8.3).

In Chora it is also possible to derive an auto object variant from another auto object variant. This results in very powerful instrument for the development of multivariant GUI applications.

Within the Embedded Wizard IDE auto object variants appear as auto object bricks extended by a small icon in the Unit Composer window:

The attributes and properties of the auto object variant can be set directly in the Inspector window, when the appropriate variant brick has been previously selected:




## 4.9 Comments

All constants, resources, enumerations, sets, inline code definitions, classes and their members may be commented. Such comments are preceded by a double slash `//` and extend to the end of the line of code:

```
// This is the base class for all Menus
class Menu : Core::Group
{
  // Caption contains the string to draw in the title
  // bar of the Menu.
  property string Caption;
  ...
}
```

Unlike in other programming languages, the comments are not ignored or simply bypassed, but are read by the Chora compiler and output in the generated ANSI C code. From the standpoint of Chora, comments are fully valid syntactic elements. Therefore it is not allowed to place a comment just anywhere in a unit file. This excerpt from an erroneous unit file demonstrates this:

```
class Menu : Core::Group
{
  object Sample::Item Item1
  {
    // This is a misplaced comment line because the
    // preset can't be commented.
    preset Caption = "Sound Settings";
  }
  ...
}
```

A comment must always directly precede the definition to which it applies. Comments are very useful to document the functionality of classes, methods, properties, etc.. Embedded Wizard uses the content of comments to give you a short description of members, which have been selected in the Composer window of the Embedded Wizard application. This description appears below the Inspector window. For more details see 'Embedded Wizard User Manual'.

Additionally Embedded Wizard is able to generate a documentation file with the description of your project including the class hierarchy, declaration of classes, methods, properties, constants, resources, etc. This automatic generated documentation file can then be used or distributed as 'software reference manual' of your project. Further details to comments and their syntax can be found in "Comments" (chapter 12).

## 4.10 Example

The following example shall demonstrate the structure of a very simple unit file. The unit consists of a bitmap resource and a class. At runtime, an object of this class will be able to output the contents of the bitmap resource on the screen. This example is based on the Mosaic class library:

```
$version 5.0

// The Icon1 bitmap contains a picture of a small\
// speaker.
resource Resources::Bitmap Icon1
{
  attr bitmapfile FileName = ..\icons\icon1.png;
}

// The Speaker class will show the speaker picture\
// on the screen.
$output = true
class Speaker : Core::Group
{
  // The Image object to view the Icon1 bitmap\
  // resource.
  object Views::Image Image
  {
    preset Bounds  = <0,0,64,64>;
    preset Bitmap  = Sample::Icon1;
    preset Visible = true;
  }

  inherited property Visible = true;

  // Respond to KeyDown events and toggle the visible\
  // state of this group.
  inherited method HandleEvent()
  {
    var Core::KeyEvent keyEvent = (Core::KeyEvent)aEvent;

    // Is this a KeyDown event?
    if (( keyEvent != null ) && keyEvent.Down )
    {
      Visible = !Visible;
      return this;
    }

    // Sign this event as unhandled ...
    return null;
  }
}
```


Copyright TARA Systems GmbH

# 5 Statements

The statements of a Chora program control the flow of program execution. The programmer uses the statements to define the logic of a method. In Chora, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, etc. Here is a brief overview of supported Chora statements:

- Declaration statement → Allows the programmer to declare local variables and arrays within the body of a method. Locally declared variables and arrays serve to store intermediate results when executing the logic of a method.
- Expression statement → Determines the value of an expression. The expression consists of a sequence of operands and operators.
- Empty statement → The empty statement does nothing. It is useful when the syntax of the language expects a statement but no expression evaluation. It consists of a semicolon '`;`'.
- Compound statement → A compound statement (also called a 'block') typically appears as the body of another statement, such as the `if` selection statement. A compound statement combines several statements into a single one.
- `if` selection statement → The `if` selection statement provides a means of conditionally executing sections of code. The statement controls conditional branching. The body of the `if` selection statement is executed if the value of the condition is true.
- `switch` selection statement → The `switch` selection statement helps control complex conditional and branching operations. The `switch` selection statement transfers control to a statement within its body depending on the expression in the `switch` statement.
- `for` iteration statements → The `for` iteration statement lets you repeat an other statement a specified number of times. The body of a `for` iteration statement is executed zero or more times until a condition becomes false. You can use additional expressions within the `for` iteration statement to initialize and change values during the `for` iteration statement's execution.
- `while` iteration statement → The `while` iteration statement lets you repeat an other statement a specified number of times. The body of a `while` iteration statement is executed zero or more times until a condition becomes false.
- `return` statement → The `return` statement terminates the execution of a method and returns control to the calling method. Execution resumes in the calling method at the point immediately following the call. A `return` statement can also return a value to the calling method.
- `signal` statement → The `signal` statement is for sending signals to a selected slot of an object. Together with the slot method, this command forms a simple method for communicating between objects.


Copyright TARA Systems GmbH

- `postsignal` statement → The `postsignal` statement records a pending signal to a selected slot of an object. The signal is delivered always immediately before the screen update is performed. Unlike the ordinary `signal` statement, `postsignal` is suited for deferred code execution, for example to finalize some initialization steps.
- `idlesignal` statement → The `idlesignal` statement records a pending signal to a selected slot of an object. The signal is delivered always after the screen update is performed and no user inputs are waiting for execution. Unlike the ordinary `signal` statement, `idlesignal` is suited for deferred code execution and for performing of idle tasks.
- `throw` exception statement → The `throw` statement causes an exception to be thrown. The result is an immediate transfer of control to the error handler (a global panic function within the Runtime Environment). Exceptions are non-recoverable errors that occur when executing a method. If an exception is thrown at runtime, the program can't be executed further.
- `trace` debug statement → The `trace` debug statement is only for test purposes. The programmer can use this command to evaluate an expression at runtime and to send the result to the debug console.
- `tracestack` debug statement → The `tracestack` debug statement is only for test purposes in the prototyping environment of Embedded Wizard. The programmer can use this command to evaluate the calling stack at runtime.
- `native` statement → The `native` statement allows the programmer to insert native code, for example calls to 'C' functions, directly into the body of a method. In this manner, the Chora statements and the native statements can be mixed together within one and the same method. The `native` statement is for the code generation only - the Embedded Wizard is not able to interpret the enclosed native code sequences.
- `attachobserver`, `detachobserver`, `notifyobservers` statements → These statements provide a powerful observer infrastructure for registration and delivery of notifications between objects. It serves as the fundament for the Controller-View model. In the Controller-View model, the widgets (views) and the application logic (controllers) are always kept apart. The observer infrastructure ensures, that widgets are notified automatically as soon as the affected controller has changed its state. On the other hand, user interactions on a widget cause the affected controller to execute the application logic. Usually, a controller is a simple Chora object containing several properties and the implementation of the onget/onset methods.

## 5.1 '`var`' declaration statement

A `var` statement is for declaring a local variable. The variable can then be used in the method's logic to store intermediate results, for example. The `var` statement has these two forms of syntax:

Form 1:

```
var type-expression name ;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Form 2:

```
var type-expression name = expression ;
```

The variable's data type is given explicitly with *type-expression*. Chora defines a set of data types. All of these data types may be used when defining a local variable. A detailed description of the data types is found in "Data types" (chapter 6).

Optionally, the variable can be initialized with an expression. If the expression is missing, the variable is not initialized. The Chora compiler will report an error if a non-initialized variable is used in an expression. The following example demonstrates the use of a local variable, `tmp`, to traverse a concatenated list of objects:

```
var Core::View tmp = Views;

// Hide all views ...
while ( tmp != null )
{
  tmp.Visible = false;

  // Continue with the next view
  tmp = tmp.next;
}
```

A variable's validity is limited to the block in which it was defined. Within this block, the name of the variable must be unique. Identically named variables may not be defined within one and the same block. For this reason, the following code causes a Chora compiler error message:

```
if ( a1 >= a2 )
{
  var int32 tmp = a1 + 10;

  if ( a1 == a2 )
  {
    var int32 tmp;
    ...
  }
}
```

The definition of a variable can occur anywhere in the logic of a method where statements are allowed. Thus it is possible to define a variable even in the middle of a block:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Core::View tmp   = Views;
var rect       union = <0,0,0,0>;

// Show all views and determine the union area
// occupied by all views ...
while ( tmp != null )
{
  // Show the view
  tmp.Visible = true;

  // Get the extent of the view
  var rect extent = tmp.GetExtent();

  // Build a union with the extent
  union = union | extent;

  // Continue with the next view
  tmp = tmp.next;
}
```

Note that the content of a variable is lost when the block is left. As a result, it is not possible to define global or static variables. All variables are automatically defined when the block is entered and removed when the block is left.

## 5.2 '`array`' declaration statement

An `array` statement is for declaring a local array. The array can then be used in the method's logic to store intermediate results, for example. The `array` statement uses the following syntax:

```
array type-expression name [ array-size ];
```

The array's data type is given explicitly with *`type-expression`*. Chora defines a set of data types. All of these data types may be used when defining a local array. A detailed description of the data types is found in "Data types" (chapter 6).

Unlike a variable, an array can store more than one value. To do this, the maximum size of an array must be given explicitly in its definition. The following example demonstrates the definition of an array with the name `points`. The array consists of a total of `16` elements, each of which stores a floating-point value:

```
array float points[ 16 ];
```

The elements of an array must always be initialized before use in an expression. The Chora compiler will report an error if a non-initialized array item is used in an expression. The following example demonstrates the initialization of a local array with the `0.0` value:

```
array float points[ 16 ];
var int32 inx;

// Initialize all items of the array with 0.0
for ( inx = 0; inx < 16; inx = inx + 1 )
  points[ inx ] = 0.0;

...
```

The validity of an array is limited to the block in which it was defined. Within this block, the name of the array must be unique. Identically named arrays may not be defined within one and the same block. For this reason, the following code causes a Chora compiler error message:

```
if ( a1 >= a2 )
{
  array int32 tmp[4];
  ...

  if ( a1 == a2 )
  {
    array int32 tmp[2];
    ...
  }
}
```

The definition of an array can occur anywhere in the logic of a method where statements are allowed. Thus it is possible to define an array even in the middle of a block.

Note that the content of an array is lost when the block is left. As a result, it is not possible to define global or static arrays. All arrays are automatically defined when the block is entered and removed if it is left.

Additionally it is possible to define an array consisting of more than one dimensions. For this purpose the sizes of all dimensions have to be placed between the braces and separated by commas:

```
array int32 Field[4,5];
```

The access to a multi dimensional array works in the same way as the access to a single dimensional array. The index of the required array element consists of more expressions separated by commas:

```
Field[0,2] = 1369;
```

If possible omit the usage of multi dimensional arrays because of possible changes in the future implementation of Chora. The current implementation of arrays will be then replaced in order to support dynamic and open arrays.

## 5.3 Expression statement

Expression statements cause expressions to be evaluated. No transfer of control or iteration takes place as a result of an expression statement.


Copyright TARA Systems GmbH
www.embedded-wizard.de

All expressions in an expression statement are evaluated and all side effects are completed before the next statement is executed. The most common expression statements are assignments and method calls:

```
var rect Bounds   = <0,0,200,100>;
var rect Clipping = <10,10,20,20>;

UpdateArea(( Bounds + <5,2>) & Clipping );
```

This example demonstrates how the rectangle stored in the variable `Bounds` is moved a distance of `<5,2>` pixels and then forms an intersection with a second rectangle `Clipping`. The result of this expression is passed to the method `UpdateArea()`.

## 5.4 Empty statement

An empty statement is a statement containing only a semicolon `;` it can appear wherever a statement is expected. Nothing happens when an empty statement is executed. The correct way to code an empty statement is:

```
;
```

Statements such as `while`, `for`, `if` require that an executable statement appear as the statement body. The empty statement satisfies the syntax requirement in cases that do not need a substantive statement body.

This example illustrates the empty statement:

```
// Repeat until the Condition() returns false.
while ( object.Condition())
  ;
```

## 5.5 Compound statement

A compound statement can combine other statements into a block. These individual statements are enclosed by braces `{ ... }`

```
if ( menu.Visible == true )
{
  item1.Visible = true;
  item2.Visible = true;
  item3.Visible = true;
}
```

When multiple statements are assembled into a block, this block appears as a single, complex statement — a compound statement.

Compound statements are used wherever the Chora syntax allows only a single statement yet the program must execute multiple instructions at once. This applies foremost to the `if`, `while` and `for` statements.

Additionally, a compound statement defines a range of validity for all local variables and arrays defined within its block. Outside the block, these variables and arrays lose their validity and can't be used. Note the local variable `theView`:

```
array Core::View views[ 10 ];
var int32        counter;

...

for ( counter = 9; counter > 0; counter = counter - 1 )
{
  var Core::View theView = views[ counter ];

  theView.Bounds  = <0, 0, 100, 200>;
  theView.Visible = true;
}

// The variable 'theView' is out of scope here!
theView = ...   <-- Will cause syntax error
```

If necessary, compound statements may also be nested:

```
for ( counter = 9; counter > 0; counter = counter - 1 )
{
  var Core::View theView = views[ counter ];

  if ( !theView.Visible )
  {
    theView.Bounds  = <0, 0, 100, 200>;
    theView.Visible = true;
  }
}
```

If a compound statement contains no embedded statements, it is a special case of the empty statement. This statement is simply ignored and not executed at runtime:

```
if ( menu.Visible == true )
{
  // Nothing to do here.
}
```

## 5.6 'if' selection statement

The `if` statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both. The syntax for the `if` statement has two forms:

Form 1:

```
if ( expression )
  statement
```

Form 2:

```
if ( expression )
  statement
else
  statement
```

In the first form of the syntax, if *expression* is `true`, *statement* is executed. If *expression* is `false`, *statement* is ignored. In the second form of syntax, which uses `else`, the second *statement* is executed if *expression* is `false`. With both forms, control then passes from the `if` statement to the next statement in the logic. The expression must have type `bool`, or a Chora compiler error occurs.

The following example uses the `if` selection statement to test the validity of the variable `Size`. Assuming the variable may contain only values in the range `0 ... 100`, the if selection statements serve to correct the value of `Size`:

```
var int32 Size;

...

if ( Size > 100 )
  Size = 100;
else if ( Size < 0 )
  Size = 0;
```

When nesting `if` statements and `else` clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the Chora compiler resolves ambiguities by associating each `else` with the closest `if` that lacks an `else`.

## 5.7 '`switch`' selection statement

The `switch` statement transfers control to one of several statements depending on the value of the expression. The syntax for the `switch` statement has the following form:

```
switch ( expression ) { case-statement-list }

case-statement-list :
  default : statement
  case-statement
  case-statement-list case-statement

  case-statement :
    case expression-list : statement
```

A `switch` statement consists of one expression, any number of `case` statements and an optional `default` statement. These statements help control complex conditional and branching operations, which would otherwise only be possible through the use of nested `if` / `else` statements. The following example demonstrates the use of the switch statement in evaluating user input:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Core::KeyEvent event = ...

// Handle the user input ...
switch ( event.Code )
{
  case Core::KeyCode.Exit : ExitMenu();
  case Core::KeyCode.Menu : OpenMenu();
  case Core::KeyCode.Ok   : SelectOk();

  // Otherwise the event remains unhandled
  default:
    return null;
}

// The event has been handled ...
return this;
```

Depending on the result of the expression `event.Code`, the `switch` statement automatically branches into one of the `case` statements. Each `case` statement defines a constant value for the purpose, for example `Core::KeyCode.Exit`. If the result of the `event.Code` expression agrees with one of the constant values, the appropriate case statement is executed. No other statements are executed. If, for example, the expression `event.Code` gives the result `Core::KeyCode.Menu`, the method `OpenMenu()` will be called in the corresponding `case` statement.

The optional `default` statement is used when the result of an expression does not agree with any of the `case` statements. In this case, the `switch` statement branches into the `default` statement. If the `default` statement is missing, however, no statements are executed in this case and the `switch` statement is left.

The absence of the `default` statement can in some cases lead to a logical runtime error if all of the possible results of the expression are not covered by suitable `case` statements. To make the programmer aware of this possible source of error, the absence of a `default` statement leads to a warning from the Chora compiler.

Please note that the expression in a `switch` statement and the constants in the dependent `case` statements must always be of the same data type. A combination of various data types is not allowed and leads to a syntax error from the Chora compiler:

```
var char theChar = ...

switch ( theChar )
{
  case 'A'   : ...;
  case 'B'   : ...;
  case false: ...;  // Error! Combination of 'char' and
                    // 'bool' data types.

  default:
    ...
}
```

Even the choice of allowed data types is limited to only the following: `bool`, `char`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`, user defined enumeration and `language`. No other data types may be used in the `switch` statement nor in the `case` statements. The following attempt to use the data type `string` meets with an error message from the Chora compiler:

```
switch ( CaptionText )
{
  case "Hello"  : ...; // Error: 'string' datatype is
  case "Hallo"  : ...; // not supported by the 'switch'
                       // statement!

  default:
    ...
}
```

Despite the similarity in the C, C++ and Java programming languages, the Chora version of the `switch` statement has its own peculiarity: the `case` and `default` statements do not have to end with a `break`. The `switch` statement in C, C++ and Java depends on the `break` statement to mark the end of a `case` or `default` statement. If the `break` is missing, it can happen that multiple `case` statements are executed, one after the other, until either a `break` or the end of the `switch` statement is reached. Used reasonably, this technique contains potential for some programming tricks. On the other hand, forgotten or wrongly placed `break` statements are a common source of logical runtime errors. For this reason, Chora does not allow `break` statements, and every `case` / `default` statement is neatly ended automatically.

In rare cases, it is desirable to define multiple, different conditions for a single `case` statement. If this condition is fulfilled, the `switch` statement automatically branches off to the corresponding `case` statement.

In order to define a `case` statement with several conditions, the constants it contains must be separated by commas when listed behind the `case` keyword. If the expression `event.Code` in the following example gives the value `Core::KeyCode.Ok` or `Core::KeyCode.Menu`, the method `OpenMenu()` is called in both cases:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Core::KeyEvent event = ...

// Handle the user input ...
switch ( event.Code )
{
  case Core::KeyCode.Ok,
       Core::KeyCode.Menu :
    OpenMenu();

  case Core::KeyCode.Red,
       Core::KeyCode.Yellow,
       Core::KeyCode.Blue,
       Core::KeyCode.Green :
    OpenDialog();

  // Otherwise the event remains unhandled
  default:
    return null;
}

// The event has been handled ...
return this;
```

Like all other statements, the switch statement may also be nested:

```
var uint8 bitField = ...

switch ( bitField & 0x03 )
{
  case 0x01: ...
  case 0x02:
    switch ( bitField & 0xF0 )
    {
       case 0x10: ...
       case 0x20, 0x30, 0x40: ...
       case 0x50: ...
       default: ...
    }

  default: ...
}
```

## 5.8 '`for`' iteration statement

The `for` iteration statement executes an expression and a statement repeatedly until the value of the expression is `false`. The syntax for the `for` statement has two forms:

Form 1:

```
for ( ; condition; loop-expression )
  statement
```

Form 2:

```
for ( init-expression; condition; loop-expression )
  statement
```

There is no restriction on the type of *init-expression* and *loop-expression*. The *condition* expression must be of type `bool`, otherwise the Chora compiler returns an error. Execution of a `for` statement proceeds as follows:

1. The *init-expression* (if any) is evaluated. This specifies the initialization for the loop. The *init-expression* is evaluated only once, at the time the `for` iteration statement is begun.
2. The *condition* expression is evaluated. It is evaluated before each iteration.
3. If the condition is `true`, the *statement* is executed and then the *loop-expression* is evaluated. The *loop-expression* is evaluated after each iteration. The process then starts again with the evaluation of *condition* expression (step 2.).
4. If the condition is `false`, execution of the `for` statement terminates and control passes to the next statement in the program. No *loop-expression* is evaluated.

The following example uses a `for` loop to compute the average of all five elements of the array `numbers`:

```
var int32   inx;
array int32 numbers[5];
var int32   average;

...

for ( inx = 0; inx < 5; inx = inx + 1 )
  sum = sum + numbers[ inx ];

average = sum / 5;
```

## 5.9 '`while`' iteration statement

The `while` statement executes an expression and a statement repeatedly until the value of the expression is `false`. The syntax for the while statement:

```
while ( expression )
  statement
```

The *expression* must be of type `bool`, otherwise a Chora compiler error occurs. Execution proceeds as follows:

1. The *expression* is evaluated.
2. If the *expression* is initially `false`, the body of the `while` statement is never executed, and control passes from the `while` statement to the next statement in the logic.
3. If the *expression* is `true`, the body of the statement is executed and the process is repeated beginning at step 1.


Copyright TARA Systems GmbH
www.embedded-wizard.de

The following example uses the `while` statement to copy all `5` elements of array `list1[]` into array `list2[]`. The `while` statement is run through repeatedly until the condition `inx < 5` is no longer fulfilled → that is, until all `5` elements of the array are copied:

```
var int32   inx = 0;
array color list1[5];
array color list2[5];

...

// Copy all 5 entries of array list1[] to the
// array list2[]
while ( inx < 5 )
{
  list2[ inx ] = list1[ inx ];
  inx = inx + 1;
}
```

## 5.10 '`return`' statement

A `return` statement returns control to the calling method. The statement can also return a value, so that the caller can evaluate it. The syntax for the `return` statement has two forms:

Form 1:

```
return ;
```

Form 2:

```
return expression ;
```

The `return` statement, used without an expression, may be used only in a method that can't return values. → The method was defined using the return value type `void`. Using the `return` statement, execution of the method is aborted immediately and the control returns to the caller.

The `return` statement, used with an *`expression`*, is used to return the result of the *`expression`* to the caller. This form of the `return` statement may only be used in a method that must return a value. → The definition of the method contains a return data type that is not `void`.

The Chora compiler reports an error if the `return` statement used does not match the definition of the method. If the method was defined using the return data type `void`, only the first form of the return statement may be used without an expression. If the method is to return a value, then the method must be closed using the second form of the `return` statement.

The following example uses the `return` statement to prematurely stop execution of the method if `value` has not changed:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
// Nothing to do? If no -> return immediately
if ( value == oldValue )
  return;

oldValue = value;
...
```

## 5.11 '`signal`' statement

A `signal` statement sends a signal to an object in order to notify that object of an event. The signal is sent to a slot of an object, using the corresponding slot method. The `signal` statement has the following syntax:

```
signal expression ;
```

The *`expression`* in the `signal` statement must result in data type `slot`, otherwise the Chora compiler will indicate an error. The following example demonstrates the use of the `signal` statement to send a signal to the `OnAction` slot of the object `theObject`:

```
var slot theSlot = theObject.OnAction;

// send a signal to the OnAction slot of the theObject
// object.
signal theSlot;
```

Note that in Chora it is not absolutely necessary to react to a signal that is sent. If the expression in the `signal` statement results in the value `null`, then no signal is sent and program execution continues without resulting in any runtime errors:

```
var slot theSlot = null;

// The variable theSlot can still keep null sometimes
if ( visible )
  theSlot = theObject.OnAction;

// Try to send the signal.
signal theSlot;
```

When a signal is sent, the identity of the sender object is passed on to the slot method of the receiver object in a hidden argument `sender`. Within the slot method, the source of the signal can thus be determined. Except for the identity of the sender, a signal delivers no further data. Signals exist only for the sake of notification that "something has changed". Additional details in "slot methods" (chapter 4.1.5.4).

Additionally, the statement `postsignal` and `idlesignal` can be used to send a pending signal after a short delay. Further details on pending signals are found in "'postsignal' statement" (chapter 5.12) and "'idlesignal' statement" (chapter 5.13)


Copyright TARA Systems GmbH
www.embedded-wizard.de

## 5.12 'postsignal' statement

A postsignal statement records a pending signal to an object for a deferred delivery. The signal is delivered immediately before the screen update is performed. When the signal is delivered, it is sent to a slot of an object, using the corresponding slot method. The postignal statement has the following syntax:

```
postsignal expression ;
```

The *expression* in the postsignal statement must result in data type slot, otherwise the Chora compiler will indicate an error. The following example demonstrates the use of the postsignal statement to record a pending signal to the OnAction slot of the object theObject:

```
var slot theSlot = theObject.OnAction;

// Record a signal to the OnAction slot of the theObject
// object.
postsignal theSlot;
```

Note that in Chora it is not absolutely necessary to react to a signal that is sent. If the expression in the postsignal statement results in the value null, then no pending signal is recorded and program execution continues without resulting in any runtime errors:

```
var slot theSlot = null;

// The variable theSlot can still keep null sometimes
if ( visible )
  theSlot = theObject.OnAction;

// Try to record the signal.
postsignal theSlot;
```

The pending signals are delivered strictly in the order, in which the corresponding postsignal statements are executed. Multiple signals to the same slot will be always merged together and delivered as a single signal. If a signal is already pending for the delivery, the postsignal statement does not record it twice - it moves the recorded signal to the end of the list of pending signals only. The example below demonstrates these aspects:

```
// Record signals to the slots OnAction1, OnAction2 and
// OnAction3.
postsignal theObject.OnAction1;
postsignal theObject.OnAction2;
postsignal theObject.OnAction3;

// Since OnAction1 already waits for the delivery, the
// following statement does not record the signal twice.
// It only changes the order of the signal delivery.
postsignal theObject.OnAction1;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Due to the last `postsignal` statement, `OnAction1` is moved to the end of the list of pending signals. When executed, the signals are delivered in the following order: `OnAction2`, `OnAction3`, `OnAction1`.

Please note, already delivered signals can't be 'posted' again as long as the screen update is not performed. In this case the `postsignal` statement fails and a runtime debug warning is displayed. This limitation is important in order to avoid an endless delivery of signals. The following example demonstrates it:

```
slot OnAction
{
  ... do something here ...

  // The following statement fails, because OnAction
  // has already received the pending signal. A debug
  // warning will appear on the debug console.
  postsignal OnAction;
}
```

Unlike the ordinary `signal` statement, `postsignal` does not pass the sender of the signal to the affected slot method. The hidden argument `sender` is always `null`, if the slot method is invoked due to a pending signal. For more details see "'signal' statement" (chapter 5.11).

Unlike the `idlesignal` statement, the delivery of 'posted' pending signals is forced always before the screen update is performed. For more details see "'idlesignal' statement" (chapter 5.13).

Please note, that pending signals do affect the garbage collection. As long as an object is waiting for a signal, the object is still reachable and will not be reclaimed by the Garbage Collector. For more details see "Garbage collection" (chapter 11).

Pending signals are very useful, if deferred code execution is necessary. For example, during the initialization of the GUI application, some initialization steps may be finalized later – still before the screen update is performed. To force code execution after the screen update is done, the statement `idlesignal` should be used. For more details regarding the slot methods see "slot methods" (chapter 4.1.5.4).

## 5.13 'idlesignal' statement

An `idlesignal` statement records a pending signal to an object for a deferred delivery. The signal is delivered as soon as possible after the screen update is performed and no user inputs are waiting for the execution. When the signal is delivered, it is sent to a slot of an object, using the corresponding slot method. The `idlesignal` statement has the following syntax:

```
idlesignal expression ;
```

The *expression* in the `idlesignal` statement must result in data type `slot`, otherwise the Chora compiler will indicate an error. The following example demonstrates the use of the `idlesignal` statement to record a pending signal to the `OnAction` slot of the object `theObject`:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var slot theSlot = theObject.OnAction;

// Record a signal to the OnAction slot of the theObject
// object.
idlesignal theSlot;
```

Note that in Chora it is not absolutely necessary to react to a signal that is sent. If the expression in the `idlesignal` statement results in the value `null`, then no pending signal is recorded and program execution continues without resulting in any runtime errors:

```
var slot theSlot = null;

// The variable theSlot can still keep null sometimes
if ( visible )
  theSlot = theObject.OnAction;

// Try to record the signal.
idlesignal theSlot;
```

The pending signals are delivered strictly in the order, in which the corresponding `idlesignal` statements are executed. Multiple signals to the same slot will be always merged together and delivered as a single signal. If a signal is already pending for the delivery, the `idlesignal` statement does not record it twice - it moves the recorded signal to the end of the list of pending signals only. The example below demonstrates these aspects:

```
// Record signals to the slots OnAction1, OnAction2 and
// OnAction3.
idlesignal theObject.OnAction1;
idlesignal theObject.OnAction2;
idlesignal theObject.OnAction3;

// Since OnAction1 already waits for the delivery, the
// following statement does not record the signal twice.
// It only changes the order of the signal delivery.
idlesignal theObject.OnAction1;
```

Due to the last `idlesignal` statement, `OnAction1` is moved to the end of the list of pending signals. When executed, the signals are delivered in the following order: `OnAction2`, `OnAction3`, `OnAction1`.

Unlike the ordinary `signal` statement, `idlesignal` does not pass the sender of the signal to the affected slot method. The hidden argument `sender` is always `null`, if the slot method is invoked due to a pending signal. For more details see "'signal' statement" (chapter 5.11).

Unlike the `postsignal` statement, the delivery of idle signals is performed always after the screen update is done and no user inputs wait for the execution. For more details see "'postsignal' statement" (chapter 5.12).

Please note, that pending signals do affect the garbage collection. As long as an object is waiting for a signal, the object is still reachable and will not be reclaimed by the Garbage Collector. For more details see "Garbage collection" (chapter 11).


Copyright TARA Systems GmbH
www.embedded-wizard.de

Idle signals are very useful to perform idle (background) tasks. An idle task should always be executed in several small steps. In this manner the GUI application can continue responding to the user inputs while the idle task performs its job step by step in the background. For example, the time-consuming decoding of a JPG picture can be done by an idle task in several steps. Each time an idle signal is delivered, the next decoding step is executed:

```
slot MyIdleTask
{
  // Perform the next decoding step.
  Decode();

  // Anything more to do next time? If yes, request
  // one more idle signal.
  if ( AnythingMoreToDo())
    idlesignal MyIdleTask;

  // ... otherwise the task is done.
  else
    ...
}
```

To start the idle task, an ignition signal should be sent to the task's slot method. Afterwards the task's slot method will keep itself running by executing the `idlesignal` statement.

The `idlesignal` statement can be also used, if during the initialization of the GUI application, some initialization steps may be finalized later – but after the screen update is performed. To force code execution before the screen update is done, the statement `postsignal` should be used. For more details regarding the slot methods see "slot methods" (chapter 4.1.5.4).

## 5.14 '`throw`' exception statement

A `throw` statement is used when a fatal, non-recoverable error has occurred such that continued execution of a program is not possible. At runtime, the `throw` statement passes control to an error handler (panic function), which must undertake all necessary steps to restart the system. The `throw` statement has the following syntax:

```
throw expression ;
```

The *`expression`* in the `throw` statement must result in data type string (character string), otherwise the Chora compiler will indicate an error. This expression should contain a brief description of the cause of the error → it can, for example, be displayed on the debug console for debugging purposes. The following example demonstrates the use of the `throw` statement following an invalid object:

```
// The view has to belong to a group!!!
if ( view.group == null )
  throw "Invalid views owner";
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Note that the `throw` statement is suited only to fatal error situations. Errors that may occur legally at runtime must be handled differently.

## 5.15 '`trace`' debug statement

The `trace` statement has the following syntax:

```
trace expression [ , expression ... ] ;
```

A `trace` statement expects one or more expressions that are evaluated when the statement is executed. The results of these expressions are converted to text and output to the debugging console. This makes it easier to search for bugs in a Chora program.

The following example demonstrates how the information on the view objects can be logged on every pass through the `while` loop:

```
var Core::View theView = ...;

// Repeat until all views are evaluated.
while ( theView != null )
{
  trace theView, theView.Visible, theView.Bounds;

  // Continue with the next view object
  theView = theView.next;
}
```

## 5.16 '`tracestack`' debug statement

The `tracestack` statement has the following syntax:

```
tracestack ;
```

A `tracestack` statement does not expect any operands. This statement outputs the names of the methods, which are currently executing in the prototyping environment, so it is easer to find out where a method has been called from. Additionally the arguments and the local variables used within the methods are displayed.

Note, this statement works in the prototyping environment of the Embedded Wizard only. During the code generation, the statement is replaced by an empty statement `;`.

## 5.17 '`native`' statement

The `native` statement inserts a user defined native code block into the body of the method. The syntax for the `native` statement has two forms:

Form 1:

```
native
{
  native-code-block
}
```

Form 2:

```
native ( identifier [ , identifier ... ] )
{
  native-code-block
}
```

The `native` statement allows the mixing of the platform independent Chora code with the target specific code. During the code generation the *native-code-block* is taken over without any modifications into the body of the generated method. The native code block is always enclosed by two braces `{}`. For example, the following method contains the Chora code and a native statement with two target specific 'C' function calls:

```
// At first use Chora to create a new object.
var Menu::Item item = new Menu::Item;

item.Caption  = "Volume";
item.OnAction = OnAction;

// Now force some 'C' function calls, which are strict
// target specific.
native
{
  dispatch_command( CMD_UPDATE_TUNER );
  send_i2c_command();
}

// Finally display the previously created menu item.
// This can and should be expressed in the platform
// independent Chora language.
Add( item, 0 );
```

The Chora compiler ensures, that all Chora statements are properly translated into the corresponding target specific code. In contrast to it, all code blocks, which are enclosed within a `native` statement, will be taken over into the generated code without any evaluation or verification - the Chora compiler is not able to detect any errors within the native code block!

Please note the usage of the braces `{}` to enclose the native code block. These braces belong to the `native` statement and are not taken over into the generated code. If the enclosed code block also contains `{}` braces, it is very important to ensure, that each opened `{` brace is always balanced with a corresponding closed `}` brace, otherwise the end of the native statement can't be recognized by the Chora compiler! To insert a unbalanced brace, an optional backslash sign `\` in front of the brace should be used:

```
native
{
  if ( ... )
  {
    printf( "A unbalanced brace \{" );
  }
```


Copyright TARA Systems GmbH

```
    ...
} // The end of the native statement
```

The `native` statement is available during the code generation only. In the prototyping environment of the Embedded Wizard no `native` statements are tolerated - the Chora interpreter is not able to execute the enclosed target specific code, so an error is reported. In order to distinguish between different target platforms and the prototyping environment, it is necessary to use the Chora preprocessor directives:

```
$if $prototyper
  ...
  Some Chora code to execute in the prototyping
  environment only.
  ...
$else
  native
  {
    dispatch_command( CMD_UPDATE_TUNER );
    send_i2c_command();
  }
$endif
```

By using the `$if`, `$else`, `$elseif` and `$endif` Chora preprocessor directives, part of the code block, depending on the target platform, can be ignored or taken over. If the condition in an `$if` or `$elseif` directive is fulfilled (is `true`), the following part of the code block is taken over. If the condition is not fulfilled, the code block is ignored. Details of the Chora preprocessor are found in "Chora preprocessor" (chapter 8).

Usually, the code enclosed in the `native` statement needs an access to the environment in which it is implemented - it should be able to read and modify local variables and method parameters. To ensure, the desired variables and parameters are accessible from the native code, their names should be placed in the declaration part of the `native` statement:

```
method Example::PageNr GetPageNr( var int32 aIndex )
{
  // Declare some variables to share between Chora
  // and the native code.
  var int32 nr1;
  var int32 nr2;

  // In case of the prototyper, initialize the both
  // local variables properly.
  $if $prototyper
    nr1 = ...
    nr2 = ...

  // In case of the target system, call a function,
  // which is then responsible for the initialization
  // of the local variables.
  $else
    native ( nr1, nr2, aIndex )
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
    {
      get_page_nr( aIndex, &nr1, &nr2 );
    }
  $endif

  // Now, the local variables are initialized and can
  // be stored in a new PageNr object.
  var Example::PageNr pageNr = new Example::PageNr;

  pageNr.Nr1 = nr1;
  pageNr.Nr2 = nr2;

  return pageNr;
}
```

Beside the local variables and method parameters, the code enclosed within the `native` statement may also access members of `this` object. Unlike the local variables and method parameters, the access to these members presumes a deep understanding, how Chora statements and expressions are translated into the language of the appropriate target system:

```
native ( this )
{
  if ( this->Super3.group != 0 )
    ...
}
```

We do not recommend to access object members directly from the native code – instead of this, local variables should be used. In this manner the native code is more unsusceptible to future changes in the Chora compiler.

Beside the `native` statement, Chora also supports native methods - see "Native methods" (chapter 4.1.5.1) - where the native method may contain the target specific code only – the mixing with the Chora code is not possible. Therefore, we recommend the using of the `native` statement instead of the obsolete native methods.

## 5.18 'attachobserver' statement

The `attachobserver` statement registers a slot method as an observer for notifications triggered for a given property or object. The syntax for the `attachobserver` statement has three forms:

Form 1:

```
attachobserver slot-method , object [ , id ];
```

Form 2:

```
attachobserver slot-method , property-reference [ , id ];
```

Form 3:

```
attachobserver slot-method , null [ , id ];
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

These three forms differ in the type of the affected source. In the first case, the slot method is attached as an observer, which will receive notifications triggered for an object *object*. In the second case the observer will receive notifications triggered for a property specified in the expression *property-reference*. The third form allows the registration of a global, object and property independent observer.

```
attachobserver update, ^controller.VolumeLevel;
```

The example above demonstrate the usage of the statement to register the slot method `update` as an observer for notifications triggered for the property `VolumeLevel` of the object `controller`. As soon as the property has been triggered, the attached slot method will receive a post signal automatically – see "'postsignal' statement" (chapter 5.12).

The notifications are triggered by the statement `notifyobservers` – see "'notifyobservers' statement" (chapter 5.20). For example, to trigger the property from the above example, the following statement is used:

```
notifyobservers ^controller.VolumeLevel;
```

As expected, this operation causes the Chora Runtime Environment to deliver a signal to the previously registered slot method `update`. In the case, there have been several methods attached to the triggered property; all affected slot methods will receive the signals.

The `attachobserver` statement does not really connect the slot method with the given property. It uses the property as a unique ID only. This ensures, that different observer groups may exist at the same time – without any conflicts. The following example demonstrates it:

```
// First attach two observers to two different
// properties.
attachobserver updateThis, ^firstController.VolumeLevel;
attachobserver updateThat, ^secondController.VolumeLevel;

...

// Then notify the observers attached to the property
// 'secondController.VolumeLevel'. In response to this
// the slot method 'updateThat' will receive a signal -
// 'updateThis' will not.
notifyobservers ^secondController.VolumeLevel;
```

If the observer is no more needed, the statement `detachobserver` can be used. After the following operation the slot method `update` is detached from the property – it is not able to receive any notifications triggered for the `VolumeLevel` property:

```
detachobserver update, ^controller.VolumeLevel;
```

For more details about `detachobserver` see "'detachobserver' statement" (chaper 5.19).


Copyright TARA Systems GmbH
www.embedded-wizard.de

In the same manner, as it has been demonstrated in the examples above, a slot method can also be attached as observer for notifications triggered for an object. In this case the `attachobserver` statement expects an expression, which results in an object:

```
attachobserver update, controller;
```

After this operation the `update` slot method is able to receive post signals - each time, the `notifyobservers` statement triggers the object `controller`:

```
notifyobservers controller;
```

If one of the expressions specified in the `attachobserver` statement results in a `null` value, the observer is not registered and the program execution is continued without any error messages:

```
var slot   theSlot   = null;
var object theObject = null;

attachobserver theSlot, theObject;
```

This should not be mixed with the third form of the `attachobserver` statement, where the `null` value is passed explicitly as the second operand of the statement. This form of the statement is designed especially for the registration of global, object and property independent observers:

```
attachobserver update, null;
```

As the example above demonstrates, the slot method `update` has been registered as a global observer. This observer can then be notified in the following manner:

```
notifyobservers null;
```

In order to provide several, independent groups of global observers, the optional *id* operand has to be used. The `attachobserver` statement expects here a `uint32` expression. The result of this expression allows an additional subdivision of observer groups. In the following example, the slot methods are attached as observers for two different global notifiers. The identity of these notifiers is specified by the last operands:

```
// First register two observers
attachobserver updateThis, null, 1;
attachobserver updateThat, null, 2;

...

// Then notify the observers with the ID 2. In response
// to this, the slot method 'updateThat' will receive a
// signal - 'updateThis' will not.
notifyobservers null, 2;
```

The optional *id* operand can also be used together with an object or a property reference. In this case, several, independent groups of observers may exists – all attached to one and the same object or property. The determination of the affected group is performed on the basis of the optional *id* operand:

```
attachobserver update, controller, 1369;
```

If the optional operand is missed, the value `0` (zero) is assumed automatically:

```
attachobserver update, controller;

...

// It will work! The slot method 'update' will receive
// a signal.
notifyobservers controller, 0;
```

The concept of observers extends the Chora concept of signals. It provides a fundament for the development of applications, which use the Controller-View model.

## 5.19 'detachobserver' statement

The `detachobserver` statement de-registers a slot method from the given property or object. The syntax for the `detachobserver` statement has three forms:

Form 1:

```
detachobserver slot-method , object [ , id ];
```

Form 2:

```
detachobserver slot-method , property-reference [ , id ];
```

Form 3:

```
detachobserver slot-method , null [ , id ];
```

These three forms differ in the type of the affected source. In the first case, the slot method is detached from the object *`object`*. In the second case the observer will be detached from the property specified in the expression *`property-reference`*. The third form allows the de-registration of a global, object and property independent observer.

```
detachobserver update, ^controller.VolumeLevel;
```

The example above demonstrate the usage of the statement to de-register the slot method `update` from the property `VolumeLevel` of the object `controller`. As soon as the observer has been detached, it is not able to receive any notifications anymore.

If the detached observer has been notified shortly before, the pending signal will always be delivered, even the observer has been detached in the meantime:

```
// Notify all affected observers - after this signals
// are waiting for the delivery
notifyobservers ^controller.VolumeLevel;

...

detachobserver update, ^controller.VolumeLevel;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
...

// Although the observer has been detached, the signal
// will be delivered
```

In the same manner, as it has been demonstrated in the examples above, a slot method can also be detached from an object. In this case the `detachobserver` statement expects an expression, which results in an object:

```
detachobserver update, controller;
```

If one of the expressions specified in the `detachobserver` statement results in a `null` value, the observer is not de-registered and the program execution is continued without any error messages:

```
var slot   theSlot   = null;
var object theObject = null;

detachobserver theSlot, theObject;
```

This should not be mixed with the third form of the `detachobserver` statement, where the `null` value is passed explicitly as the second operand of the statement. This form of the statement is designed especially for the de-registration of global, object and property independent observers:

```
detachobserver update, null;
```

In order to provide several, independent groups of global observers, the optional *id* operand has to be used. The `detachobserver` statement expects here a `uint32` expression. The result of this expression allows an additional subdivision of observer groups. In the following example, the slot methods are detached from two different global notifiers. The identity of these notifiers is specified by the last operands:

```
detachobserver updateThis, null, 1;
detachobserver updateThat, null, 2;
```

The optional *id* operand can also be used together with an object or a property reference. In this case, several, independent groups of observers may exists – all attached to one and the same object or property. The determination of the affected group is performed on the basis of the optional *id* operand:

```
detachobserver update, controller, 1369;
```

If the optional operand is missed, the value `0` (zero) is assumed automatically.

Observers, which have not been detached 'manually', are discarded by the Garbage Collector automatically, as soon one the affected objects do not exist anymore.

## 5.20 '`notifyobservers`' statement

The `notifyobservers` statement triggers the given property or object in order to notify all observers, which have been previously attached to the affected property or object. The syntax for the `notifyobservers` statement has three forms:

Form 1:

```
notifyobservers object [ , id ];
```

Form 2:

```
notifyobservers property-reference [ , id ];
```

Form 3:

```
notifyobservers null [ , id ];
```

These three forms differ in the type of the affected source. In the first case, the statement triggers an object *object*. In the second case the property specified in the expression *property-reference* is triggered. The third form allows the notification of global, object and property independent observers. The following example notifies all observers previously registered by the property `VolumeLevel` of the object `controller`:

```
notifyobservers ^controller.VolumeLevel;
```

As soon as the property has been triggered, the attached slot methods will receive post signals automatically – see "'postsignal' statement" (chapter 5.12).

Before observers may receive notifications, the observers need to be registered. This is done by the statement `attachobserver` – see "'attachobserver' statement" (chapter 5.18). The following example demonstrates, how a slot method `update` is registered as an observer:

```
attachobserver update, ^controller.VolumeLevel;
```

If the observer is no more needed, the statement `detachobserver` can be used. After the following operation the slot method `update` is detached from the property – it is not able to receive any notifications triggered for the `VolumeLevel` property:

```
detachobserver update, ^controller.VolumeLevel;
```

For more details about `detachobserver` see "'detachobserver' statement" (chaper 5.19).

If the expression specified in the `notifyobservers` statement results in a `null` value, no observers are notified and the program execution is continued without any error messages:

```
var object theObject = null;

notifyobservers theObject;
```

This should not be mixed with the third form of the `notifyobservers` statement, where the `null` value is passed explicitly as the first operand of the statement. This form of the statement is designed especially for the notification of global, object and property independent observers:

```
notifyobservers null;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

In order to provide several, independent groups of global observers, the optional *id* operand has to be used. The `notifyobservers` statement expects here a `uint32` expression. The result of this expression allows an additional subdivision of observer groups. In the following example, the slot methods are attached as observers for two different global notifiers. The identity of these notifiers is specified by the last operands:

```
// First register two observers
attachobserver updateThis, null, 1;
attachobserver updateThat, null, 2;

...

// Then notify the observers with the ID 2. In response
// to this, the slot method 'updateThat' will receive a
// signal - 'updateThis' will not.
notifyobservers null, 2;
```

The optional *id* operand can also be used together with an object or a property reference. In this case, several, independent groups of observers may exists – all attached to one and the same object or property. The determination of the affected group is performed on the basis of the optional *id* operand:

```
notifyobservers controller, 1369;
```

If the optional operand is missed, the value `0` (zero) is assumed automatically.


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 6 Data types

Knowledge of the data types is of great importance when defining constants, variables or properties. Each definition requires explicit information regarding the desired data type. With this information, the Chora compiler is able to recognize what content a variable will have in the target system, and can recognize when operations are performed on a variable that are invalid for its data type.

The choice of supported data types fits the requirements for development of GUIs. For this reason, the language supports three different kinds of data types:

- Instant data types → The instant data types are an integral part of the Chora programming language and can be used in any program without additional definitions. They can't, however, be changed.
- User-defined data types → The user-defined data types include classes, enumerations and sets. The programmer may define an arbitrary number of data types to use in a Chora program.
- Reference data types → Allow the building of links to object's properties in order to access these properties. In this way, it is possible for one object to use a reference to read or write the property of another object without needing to know anything about the existence of that object.

## 6.1 Instant data types

The instant data types are an integral part of the Chora programming language. The choice of instant data types is attributable to the application field (development of 2-dimensional graphical user interfaces). The following instant data types and their literal notation are supported:

| Data type | Description |
|---|---|
| `int8` | Signed 8 bit integer values in the range `-128` ... `+127`. |
| `int16` | Signed 16 bit integer values in the range `-32768` ... `+32767`. |
| `int32` | Signed 32 bit integer values in the range `-2147483648` ... `+2147483647`. |
| `uint8` | Unsigned 8 bit integer values in the range `0` ... `255`.<br>Unsigned integers may be expressed in either decimal or hexadecimal notation. A number in hexadecimal notation begins with the prefix `0x`:<br>`var uint8 number = 0x12;` |
| `uint16` | Unsigned 16 bit integer values in the range `0` ... `65535`.<br>Unsigned integers may be expressed in either decimal or hexadecimal notation. A number in hexadecimal notation begins with the prefix `0x`:<br>`var uint16 number = 0x12AC;` |


Copyright TARA Systems GmbH

| | |
|---|---|
| `uint32` | Unsigned 32 bit integer values in the range `0` ... `4294967295`.<br><br>Unsigned integers may be expressed in either decimal or hexadecimal notation. A number in hexadecimal notation begins with the prefix `0x`:<br><br>`var uint32 number = 0x76FF76AB;` |
| `float` | Floating-point number with single precision. The actual precision of the `float` data type can depends on the platform.<br><br>The floating-point numbers must always be expressed in decimal-point notation:<br><br>`var float angle = 130.75;` |
| `bool` | Boolean data type. Defines whether an expression is true or false.<br><br>The boolean can adopt only one of two possible values, `true` or `false`:<br><br>`var bool ItIsTrue  = true;`<br>`var bool ItIsFalse = false;` |
| `char` | A single character. Each character occupies `16` bit in memory, so it is possible to encode all known characters. Use of the worldwide character-encoding standard UNICODE is possible.<br><br>A character is always enclosed by a set of single quotes `''` . Characters that can't be entered directly from the keyboard can be expressed in hexadecimal notation by using the `\x` escape sequence:<br><br>`var char sign1 = 'S';`<br>`var char sign2 = '\xBC74';`<br><br>In this way, one is able to enter individual characters of a foreign language such as Japanese without having to depend on that keyboard or input machine. Please note, that the `\x` escape sequence always expects to be followed by 4 hexadecimal digits.<br><br>The following list shows the entire set of supported escape sequences. Each sequence always starts with a backslash sign `\`. The Chora compiler will automatically replace each found escape sequence with the corresponding character code:<br><br>`\a : 0x0007 (BEL) Alert`<br>`\b : 0x0008 (BS) Backspace`<br>`\f : 0x000C (FF) Formfeed`<br>`\n : 0x000A (LF) Newline`<br>`\r : 0x000D (CR) Carriage return`<br>`\t : 0x0009 (HT) Horizontal tab`<br>`\v : 0x000B (VT) Vertical tab`<br>`\\ : 0x005C \ Backslash`<br>`\' : 0x0027 ' Single quote`<br>`\" : 0x0022 " Double quote`<br>`\x#### : 0x#### Character in 4 hex. digit notation`<br>`\0 : 0x0000 Null Character`<br><br>For example, to express the new-line character the following escape sequence could be applied: |

```
var char theNewLineChar = '\n';
```

`string`

A string of characters. Each character occupies `16` bit in memory, so it is possible to encode all known characters. Use of the worldwide character-encoding standard UNICODE is possible.

A character string is always enclosed by a set of double quotes `""` . If the character string contains no characters, it is the so-called empty string, consisting only of two double quotes `""` :

```
var string DayOfTheWeek = "Friday";
var string EmptyString  = "";
```

If a character string must contain characters that can't be entered directly from the keyboard, these characters can be expressed in hexadecimal notation by using the `\x` escape sequence:

```
var string Text = "\x1001\x6CA9 - Text";
```

In this way, one is able to enter individual characters of a foreign language such as Japanese without having to depend on that keyboard or input machine. Please note, that the `\x` escape sequence always expects to be followed by 4 hexadecimal digits.

The following list shows the entire set of supported escape sequences. Each sequence always starts with a backslash sign `\`. The Chora compiler will automatically replace each found escape sequence with the corresponding character code:

```
\a     : 0x0007 (BEL) Alert
\b     : 0x0008 (BS)  Backspace
\f     : 0x000C (FF)  Formfeed
\n     : 0x000A (LF)  Newline
\r     : 0x000D (CR)  Carriage return
\t     : 0x0009 (HT)  Horizontal tab
\v     : 0x000B (VT)  Vertical tab
\\     : 0x005C  \    Backslash
\'     : 0x0027  '    Single quote
\"     : 0x0022  "    Double quote
\x#### : 0x####       Character in 4 hex. digit notation
\0     : 0x0000       Null Character
```

For example, to terminate a string with the new-line character the following escape sequence could be applied:

```
var string text = "Hello World!\n";
```

`color`

True-Color value in the `32` bit RGBA (red, green, blue, alpha) format.

The red, green, blue and alpha components of a color are expressed in hexadecimal notation behind a `#` (number sign):

```
var color Black       = #000000FF;
var color Yellow      = #FFFF00FF;
var color DarkBlue    = #000050FF;
var color Red         = #FF0000FF;
var color Transparent = #00000000;
```

The first two digits of a color define the red component. The green

| | |
|---|---|
| | component follows, then the blue, and finally the value of the alpha component. Each component is defined using two hexadecimal numbers in the range `00` to `FF`.<br><br>If the alpha value given is `00`, the color is transparent, independent of the red, green nor blue values. |
| `point` | Structure containing the `x, y` coordinates of a point.<br><br>The `x, y` coordinates of a point are always enclosed in angle brackets and separated by a comma:<br><br>`var point Origin = <0,0>;`<br>`var point NextPosition = <20,38>;`<br>`var point Offset = <-48,50>;`<br><br>A point stores the value of each `x` and `y` coordinate as a 32-bit signed integer. |
| `rect` | Structure containing the $(x_1,y_1)$-$(x_2,y_2)$ coordinates of a rectangle.<br><br>The coordinates of a rectangle are always enclosed in angle brackets and separated by a comma:<br><br>`var rect Bounds = <0,0,120,64>;`<br>`var rect Frame = <-10,-20,40,55>;`<br><br>A rect stores the value of each coordinate as a 32-bit signed integer. |
| `language` | Language variant data type. A variable of type `language` stores the name of a language variant. The name used for the language must be defined within the project file:<br><br>`var language currentLang = German;`<br>`var language defaultLang = Default;`<br><br>The data type `language` is used when switching between languages and in resolving language-dependent resources and constants.<br><br>The user can store the name of a language variant within a local variable, evaluate the value of this variable, use it to switch between languages, etc.<br><br>Aside from the data type `language`, there exists an identically named global built-in variable that stores the name of the currently selected language variant. Details on switching between languages can be found in "Language selection" (chapter 9). |
| `styles` | Styles set data type. A variable of type `styles` can store a list of style names. The list is enclosed in brackets and the names are separated by comma signs. An empty styles set consists of empty brackets only:<br><br>`var styles temp = [Aqua, WideScreen];`<br>`var styles empty = [];`<br><br>The data type `styles` is used in multivariant GUI applications, when at the runtime a variant of a class, constant or resource should be determined. A multivariant application implements for each style different |

| | |
|---|---|
| | appearance and behavior. This can be switched at the runtime.<br><br>The user can store the styles set within a local variable, evaluate the value of this variable, use it to switch between variants, etc.<br><br>Aside from the data type `styles`, there exists an identically named global built-in variable that stores the names of the currently selected styles. Details on switching between variants can be found in "Usage of Variants" (chapter 10). |
| `slot` | Slot method data type. This data type contains a reference to an object's slot method. Through a slot, it is possible to send simple signals to an object:<br><br>`var slot OnSelect   = Item.Select;`<br>`var slot OnActivate = null;`<br><br>`signal OnSelect; // Send a signal.`<br><br>If a signal is sent to an object's slot, the slot method corresponding to the slot is automatically executed.<br><br>The slots and signals provide a simple mechanism for communication between the objects. Details on the use of slots are described in "slot methods" (chapter 4.1.5.4) and "'signal' statement" (chapter 5.11). |
| `class` | Class data type. This data type contains a reference to a Chora class. This reference can then be used in order to create objects of this class dynamically:<br><br>`var class theClass1 = classof otherObject;`<br>`var class theClass2 = Views::Text;`<br><br>`var object theObject1 = new theClass1;`<br>`var object theObject2 = new theClass2;`<br><br>The `class` data type provides a very powerful construct, which can be used to create, for example object factories. Details on the creation of objects are described in "'new' operator" (chapter 7.6).<br><br>In order to obtain the class of an existing object, the operator `classof` is available. See "'classof' operator" (chapter 7.13). |
| `object` | Object data type. This data type represents all classes defined in a Chora program. A variable of type `object` can, therefore, store a reference to an object of any class, without the Chora compiler reporting a data type mismatch error when this variable is assigned:<br><br>`var object theObject = new Sample::Item; // Ok`<br><br>`theObject = new Audio::Device; // Ok`<br>`theObject = new Core::Timer;  // Ok`<br>`theObject = null;             // Ok`<br><br>The `object` data type is analogous to a universal, generic class, which means that it can't be used in the context of object methods, properties, variables, and so on. The Chora compiler is not capable during the |

| | |
|---|---|
| | translation of recognizing the class behind an `object`. In some cases, this class can even be changed at runtime through a assignment to the `object` variable. For this reason, any attempt to use a variable, method, property, etc. in the context of `object` leads to an error message:<br>`var object theObject = new Sample::Item;`<br>`theObject.Visible = true; // Error!`<br>Only an explicit dynamic object cast guarantees that the generic `object` data type can be casted in a particular class. After the object cast has been performed successfully, the members of the object may be accessed:<br>`var object theObject = new Sample::Item;`<br>`((Sample::Item)theObject).Visible = true; //Ok!`<br>Note that the object cast does not apply any sort of transformation to the object. Instead, it merely performs a comparison between the object's class and the desired class. If the object cast fails, the expression returns `null`, leading to a runtime error if the following security check is missing:<br>`var object theObject = new Sample::Item;`<br>`if ((Sample::Item)theObject != null )`<br>`  ((Sample::Item)theObject).Visible = true;`<br>Further information about the object cast is found in "Object cast operators" (chapter 7.4). |

**Table 6–1**

A big advantage of instant data types lies in the predefined instant operators. With the help of these operators, the programmer can construct complex expressions with `string`, `rect`, `point`, etc. operands. For example, he can combine two character `strings` into one, using the `+` operator:

```
var string Text1 = "Hello";
var string Text2 = "world";
var string Result;

Result = Text1 + " " + Text2;
```

A detailed description of the operators can be found in "Operators" (chapter 7).

## 6.2 User defined data types

User-defined data types are restricted to classes, enumerations and sets. The programmer uses these data types in definitions of variables, properties, arrays, etc. In the case of classes, objects of the relevant class can be created.

The user-defined data types are hardly different from the instant data types. The only thing the programmer needs to bear in mind is using the full name of a user-defined data type. The name of the unit must always be included and separated by a double colon '`::`' from the name of the class, enumeration or set:

```
var Sample::Item Item1 = new Sample::Item;
Item1.Caption = "Sound Settings";
```

This example demonstrates how an object of the class `Sample::Item` is created and how its property `Caption` obtains a value. Note that the name of the class is used in conjunction with the name of the unit `Sample`, where the class has been defined.

More about the definition of classes, enumerations and sets can be found in "Classes" (chapter 4.1), "Enumerations" (chapter 4.6) and "Sets" (chapter 4.7).

## 6.3 Reference data types

References allow the building of links to object's properties in order to access these properties. In this way, it is possible for one object to use a reference to read or write the property of another object without needing to know anything about the existence of that object.

A reference data type must always be derived from another, existing data type. The instant and user-defined data types are suitable for this. A reference data type is constructed using the prefix reference operator '^':

```
var ^string RefToString = null;
var ^color  RefToColor  = ^Menu.BackgroundColor;
```

This example demonstrates how references of types `^string` and `^color` are defined. The `^string` reference `RefToString` is initialized with the value `null`. This means that the reference is not yet linked to any property. The `^color` reference `RefToColor`, on the other hand, is initialized with a link to the property `BackgroundColor` of the object `Menu`. Note the use of the reference operator '^' to the left of `Menu.BackgroundColor`. This operator 'delivers' a reference to the desired property.

In order to use the reference to access the contents of the property, the reference must be dereferenced using the indirection operator '^'. Unlike the reference operator, the indirection operator is placed to the right in the expression:

```
var ^color  RefToColor  = ^Menu.BackgroundColor;
var color   NewColor    = #000000FF;
var color   OldColor;

OldColor = RefToColor^;  // read the referenced property
RefToColor^ = NewColor;  // change the property
```

Using the indirection operator, a reference is dereferenced, whereby the contents of a property may be read or written. Because every access to a property necessarily invokes a relevant onget or onset method, dereferencing of references automatically invokes the relevant method. Please note that dereferencing a `null` reference will cause a runtime error (i.e. panic). At runtime, you can test whether a reference has been correctly initialized and is not preset to `null`:

```
...
if ( RefToString != null )
  RefToString^ = "Hello world!";
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

The Chora programming language allows only simple references, meaning the references may not be nested. Creating a reference to another reference is, therefore, illegal:

```
var ^^color  RefToRefToColor  = ^^Menu.BackgroundColor;
```

In such a case, the Chora compiler will report an error message. For additional details see "Properties" (chapter 4.1.3).

# 7 Operators

Handling of the instant and user-defined data types is noticeably simplified by the operators integrated within Chora language. The programmer uses the operators to construct complex expressions consisting of different operands. Thus, the programmer can, for example, define complicated arithmetic operation. Just as easily, he can combine `string`, `point`, `rect`, `color`, etc. operators into expressions:

```
var rect Bounds   = <0,0,200,100>;
var rect Clipping = <10,10,20,20>;
var rect Result;

Result = ( Bounds + <5,2>) & Clipping;
```

This example demonstrates how the rectangle named `Bounds` is moved a distance of `<5,2>` pixels and then forms an intersection with a second rectangle `Clipping`. The result of this operation is assigned to the rectangle `Result`. The clarity and simplicity with which the programmer can form complex operations using few operators are the strengths of Chora.

The operators are fundamentally divided into the following categories:

- Instant operators → These are operators in the classical sense, such as `+`, `-`, `*`, `/`, `&`, `=`, `<=`, `!=`, ... The programmer can 'compute' using these operators, regardless of whether he is working with numbers, colors, rectangles, and so on. Each instant data type supports its own set of operators that are adapted to it. Depending on the number of operands expected, the instant operators are divided into two categories:
  - Unary instant operators → These operators work with only one operand.
  - Binary instant operators → A binary operator expects 2 operands.
- Instant cast operators → These operators enable different instant data types to be explicitly converted. Thus, the programmer can convert, for example, a floating-point number `float` to an integer `int32`. Not all type conversions are meaningful. A `rect` can't be converted to a `point`, for example. In such a case, the Chora compiler will produce an error message. It must also be noted that a type conversion can be lossy. For example, converting the signed integer `int32` to an unsigned integer `uint16` results in not only the loss of the sign, but also the loss of the upper `16` bits of the number.
- Object cast operators → An object cast operator is used to check whether an object is derived from a particular class. Due to the polymorphic nature of the objects, in most cases the Chora compiler can't predict which class the objects used in the program will have at runtime. The object cast operator allows the programmer to verify the class of an object.
- Enum and set cast operators → A special kind of cast operators, which allow the programmer to convert a number operand into an `enum` or `set` operand and vice versa. Unlike other programming languages, Chora handles the conversion very strictly and ensures, that the conversion remains legal. In this manner it avoids erroneous conversions of numbers into an `enum` or `set` operand.


Copyright TARA Systems GmbH

- `new` operator → The `new` operator is necessary to create objects of a desired class at the runtime. For example, the expression `new Views::Text` creates and initializes a new object of this `Views::Text` class. Each object is an instance of its class.
- `parentthis` operator → The `parentthis` operator determines the parent object of `this` object. The operator works for embedded objects only and allows these objects to determine their parent. If an object has not been embedded within another object, `parentthis` results in `null`.
- `classof` operator → The `classof` operator determines the class of a given object at the runtime.
- Instant constructors → An instant constructor allows the dynamic creation (construction) of instant data types at runtime of a Chora program. The instant constructor works in a way similar to a C++ or Java constructor, except that the instant constructor returns not an object, but an instant data type. For each instant data type, Chora defines a set of instant constructors. Thus, the programmer can, for example, construct a rectangle from two points, by using the instant constructor `rect()` with two point arguments: `rect(p1,p2)`.
- Instant methods → Instant methods simplify access to the instant data types. The programmer can use the instant methods to execute an operation on an instant data type. He can, for example, search a string for a particular character using the instant method `find()`. From the programmer's point of view, instant methods are used like the methods of an object.
- Instant properties → Instant properties simplify access to the instant data types. In the case of complex instant data types, such as `rect`, `point` or `color`, instant properties allow direct access to the elements of the data type. For example, the value of the alpha component in a `color` operand can be queried or changed using the instant property `alpha`. From the programmer's point of view, instant properties are used like the properties of an object.
- Index operator `[]` → The index operator may be used only in conjunction with the instant data type `string`. Using the index operator, a character can be read from or written to a string.
- Assignment operator → Using the assignment operator, the result of an expression is assigned to a variable or property.

## 7.1 Unary instant operators

A unary instant operator expects exactly one operand. For example, the unary minus operator '–' can be used in an expression to change the sign of the operand:

```
var rect  Bounds = <10,24,68,128>;
var int32 Result;

Result = -Bounds.x1;
```

In this example, the value of the Bounds.x1 operand is computed and stored with a changed sign in the variable Result.


Copyright TARA Systems GmbH
www.embedded-wizard.de

If a unary operator is used on an expression, this expression must be contained in parentheses:

```
var rect  Bounds = <10,24,68,128>;
var int32 Result;

Result = -( Bounds.x2 - Bounds.x1 + 1 );
```

If the parentheses are missing, then the operator is applied only to the first operand, with the consequence that the entire expression produces a different, in this case undesired, result.

Chora supports the following unary operators:

| Operator name | Operand data type | Description |
|---|---|---|
| + | + int8<br>+ int16<br>+ int32<br>+ uint8<br>+ uint16<br>+ uint32<br>+ float | Unary plus. The result is the unchanged value of the operand.<br>`var float a = 0.1369;`<br>`var float b = -0.1251;`<br>`var float c;`<br>`c = +a; // c = 0.1369`<br>`c = +b; // c = -0.1251` |
| – | - int8<br>- int16<br>- int32<br>- uint8<br>- uint16<br>- uint32<br>- float | Unary minus. The result is the negative of the value of the operand. The unary minus operator merely changes the sign of the operand. If the operand is unsigned (e.g. `uint16`), the operand is automatically converted into a signed data type (`int16`, `int32`).<br>`var float a = 0.1369;`<br>`var float b = -0.1251;`<br>`var float c;`<br>`c = -a; // c = -0.1369`<br>`c = -b; // c = 0.1251` |
| ~ | ~ int8<br>~ int16<br>~ int32<br>~ uint8<br>~ uint16<br>~ uint32 | Complement operator. The result is the bitwise complement of the operand. Each `0` bit in the operand is set to `1`, and each `1` bit in the operand is set to `0`.<br>This operator can be applied only to integer data types in order to be able to modify these as bitfields.<br>`var uint32 a = 0xAAAAAAAA;`<br>`var uint32 c;`<br>`c = ~a; // c = 0x55555555` |
| ! | ! bool | Logical negation. The result is the logical negation of |

| | | |
|---|---|---|
| | | the value of the operand. `false` if the operand is `true` and `true` if the operand is `false`.<br>`var bool a = true;`<br>`var bool c;`<br>`c = !a; // c = false` |

**Table 7–1**

## 7.2 Binary instant operators

A binary instant operator expects exactly two operands. For example, the string concatenation operator '+' can be used in an expression to join two character strings into one long string:

```
var string String1 = "Hello,";
var string String2 = "world!";
var string Result;

Result =  String1 + " " + String2;
```

In this example, the two strings, `String1` and `String2`, separated by a space, are joined and assigned to the variable `Result`. The result is `"Hello, world!"`.

Chora supports the following binary operators:

| Operator name | Operand data types | Description |
|---|---|---|
| + | `?int?? + ?int??`<br>`float + float` | Addition. The result is the arithmetic sum of the operands.<br>`var int32 a = 1369;`<br>`var int32 b = 1251;`<br>`var int32 c;`<br>`c = a + b; // c = 2620` |
| + | `string + string`<br>`string + char`<br>`char + string` | String concatenation. The result is a string containing a copy of the left operand followed by the copy of the right operand:<br>`var string a = "Hello ";`<br>`var string b = "World";`<br>`var string c;`<br>`c = a + b; // c = "Hello World"` |
| + | `rect + point`<br>`point + rect` | Rectangle movement. The result is the `rect` operand moved by the offset given in the `point` operand. The movement will be done in the positive direction. |


Copyright TARA Systems GmbH

| | | |
|---|---|---|
| | | `var rect  a = <10,20,30,40>;`<br>`var point b = <5,7>;`<br>`var rect  c;`<br><br>`c = a + b; // c = <15,27,35,47>` |
| + | `point + point` | Point movement. The result is the left `point` operand moved by the offset given in the right `point` operand. The movement will be done in the positivie direction.<br><br>`var point a = <10,20>;`<br>`var point b = <5,7>;`<br>`var point c;`<br><br>`c = a + b; // c = <15,27>` |
| + | `color + color` | Color addition with saturation. The addition will be done for each color component separately. In the case the addition results in a value `> 255`, the value will be adjusted automatically to `255` (=saturation).<br><br>`var color a = #10C050FF;`<br>`var color b = #225F22FF;`<br>`var color c;`<br><br>`c = a + b; // c = #32FF72FF` |
| + | `styles + styles` | Union of two styles sets. The result is a new styles set, which contains the style names from the both operands:<br><br>`var styles a = [Aqua];`<br>`var styles b = [WideScreen];`<br>`var styles c;`<br><br>`c = a + b;`<br>`// c = [Aqua,WideScreen]` |
| + | `set + set` | Union of two sets. The result is a new set, which contains the enumerators from the both operands:<br><br>`var Unit::Align a = Unit::Align[ Top ];`<br>`var Unit::Align b = Unit::Align[ Right ];`<br>`var Unit::Align c;`<br><br>`c = a + b; // c = Unit::Align[Top, Right ]` |
| – | `?int?? - ?int??`<br>`float – float`<br>`char – char` | Subtraction. The result is the arithmetic difference of the operands.<br><br>`var int32 a = 1369;`<br>`var int32 b = 1251;`<br>`var int32 c;` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | | |
|---|---|---|
| | | `c = a - b; // c = 118` |
| – | `rect - point` | Rectangle movement. The result is the `rect` operand moved by the offset given in the `point` operand. The movement will be done in the negative direction.<br>`var rect a = <10,20,30,40>;`<br>`var point b = <5,7>;`<br>`var rect c;`<br>`c = a - b; // c = <5,13,25,33>` |
| – | `point - point` | Point movement. The result is the left `point` operand moved by the offset given in the right `point` operand. The movement will be done in the negative direction.<br>`var point a = <10,20>;`<br>`var point b = <5,7>;`<br>`var point c;`<br>`c = a - b; // c = <5,13>` |
| – | `color - color` | Color subtraction with saturation. The subtraction will be done separately for each color component. In the case the subtraction results in a value `< 0`, the value will be adjusted automatically to `0` (=saturation).<br>`var color a = #10C050FF;`<br>`var color b = #225F2200;`<br>`var color c;`<br>`c = a - b; // c = #00612EFF` |
| – | `styles - styles` | Difference between two styles sets. The result is a new styles set, which contains the style names enclosed only in the left operand:<br>`var styles a = [Aqua,Win95];`<br>`var styles b = [Win95];`<br>`var styles c;`<br>`c = a - b;`<br>`// c = [Aqua]` |
| – | `set - set` | Difference between two sets. The result is a new set, which contains the enumerators enclosed only in the left operand:<br>`var Unit::Align a = Unit::Align[Top,Left];`<br>`var Unit::Align b = Unit::Align[ Left ];`<br>`var Unit::Align c;`<br>`c = a - b; // c = Unit::Align[Top ]` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | | |
|---|---|---|
| * | ?int?? * ?int??<br>float * float | Multiply. The result is the product of the both operands.<br>`var float a = 0.1369;`<br>`var float b = 0.1251;`<br>`var float c;`<br>`c = a * b; // c = 0.01713` |
| * | rect * point<br>point * rect | Rectangle inflation. The result is the `rect` operand inflated by moving its sides away from its center by the offset given in the `point` operand.<br>`var rect  a = <10,20,30,40>;`<br>`var point b = <5,7>;`<br>`var rect  c;`<br>`c = a * b; // c = <5,13,35,47>` |
| * | color * color | Alpha blending. The result is the left `color` operand alpha-blended with the right `color` operand.<br>The alpha component of the right operand decides about the intensity of the alpha blending. If this alpha component is `0` (zero), the operator returns the left color operand. If the alpha component of the right operand is `255`, the operator returns the right color operand. In all other cases, the operator returns a mix-color of the both colors.<br>`var color a = #FF000080;`<br>`var color b = #00FF0080;`<br>`var color c;`<br>`c = a * b; // c = #7F8000C0` |
| * | color * ?int??<br>?int?? * color | Alpha blending. The result is the value of the `color` operand blended with the value of the integer operand.<br>This operator applies the alpha blending algorithm on all `4` color components (red, green, blue, alpha) of the given color.<br>`var color a = #FF0000FF;`<br>`var int32 b = 0x80;`<br>`var color c;`<br>`c = a * b; // c = #80000080` |
| / | ?int?? / ?int??<br>float / float | Division. The result is the quotient.<br>`var float a = 0.1369;`<br>`var float b = 0.1251;`<br>`var float c;` |


Copyright TARA Systems GmbH

| | | |
|---|---|---|
| | | `c = a / b; // c = 1.0943` |
| % | ?int?? % ?int?? | Remainder (modulo division). The result of the operation is the reminder.<br>`var int32 a = 34;`<br>`var int32 b = 8;`<br>`var int32 c;`<br>`c = a % b; // c = 2` |
| >> | ?int?? >> ?int?? | Shift right. The result is the value of the left operand right-shifted by right operand bit positions, zero filled from the left if the left operand is unsigned operand. If the left operand is of signed type (`int8`, `int16`, `int32`), the fill from the left uses the sign bit (`0` for positive and `1` for negative).<br>`var uint32 a = 0x00AA5500;`<br>`var uint32 b = 1;`<br>`var uint32 c;`<br>`c = a >> b; // c = 0x552A80` |
| << | ?int?? << ?int?? | Shift left. The result is the value of the left operand left-shifted by right operand bit positions, zero filled from the right if necessary.<br>`var uint32 a = 0x00AA5500;`<br>`var uint32 b = 1;`<br>`var uint32 c;`<br>`c = a << b; // c = 0x154AA00` |
| & | ?int?? & ?int?? | Bitwise 'and'. The result is the bitwise 'and' of both operands.<br>`var uint32 a = 0xFF003AC0;`<br>`var uint32 b = 0xAB15FF70;`<br>`var uint32 c;`<br>`c = a & b; // c = 0xAB003A40` |
| & | rect & rect | Rectangle intersection. The result is a rectangle that represents the intersecting area of the both `rect` operands. If the intersecting area is empty, the operator returns an empty rectangle.<br>`var rect a = <10,5,110,220>;`<br>`var rect b = <-10,-50,50,75>;`<br>`var rect c;`<br>`c = a & b; // c = <10,5,50,75>` |
| & | styles & styles | Intersection of two styles sets. The result is a new |

| | | |
|---|---|---|
| | | styles set, which contains the style names enclosed in both operands:<br>`var styles a = [Aqua,Win95];`<br>`var styles b = [Win95];`<br>`var styles c;`<br>`c = a & b;`<br>`// c = [Win95]` |
| & | `set & set` | Intersection of two sets. The result is a new set, which contains the enumerators enclosed in both operands:<br>`var Unit::Align a = Unit::Align[Top,Left];`<br>`var Unit::Align b = Unit::Align[Top];`<br>`var Unit::Align c;`<br>`c = a & b; // c = Unit::Align[Top ]` |
| \| | `?int?? \| ?int??` | Bitwise inclusive 'or'. The result is the bitwise 'or' of both operands.<br>`var uint32 a = 0xFF003AC0;`<br>`var uint32 b = 0xAB15FF70;`<br>`var uint32 c;`<br>`c = a \| b; // c = 0xFF15FFF0` |
| \| | `rect \| rect` | Rectangle union. The result is a rectangle that represents the union area of the both `rect` operands. If the union area is empty, the operator returns an empty rectangle.<br>`var rect a = <10,5,110,220>;`<br>`var rect b = <-10,-50,50,75>;`<br>`var rect c;`<br>`c = a \| b;`<br>`// c = <-10,-50,110,220>`<br>If one of the operands is an empty rectangle, no union is build and the operator returns the other operand immediately.<br>`var rect a = <10,5,110,220>;`<br>`var rect b = <0,0,0,0>;`<br>`var rect c;`<br>`c = a \| b;`<br>`// c = <10,5,110,220>` |
| ^ | `?int?? ^ ?int??` | Bitwise 'xor' (exclusive 'or'). The result is the bitwise 'xor' of both operands.<br>`var uint32 a = 0xFF003AC0;`<br>`var uint32 b = 0xAB15FF70;` |

| | | |
|---|---|---|
| | | `var uint32 c;`<br><br>`c = a ^ b; // c = 0x5415C5B0` |
| ^ | styles ^ styles | Exclusive 'or' of two styles sets. The result is a new styles set, which contains the style names enclosed exclusively either in the left or in the right operand, but not in both:<br><br>`var styles a = [Aqua,Win95];`<br>`var styles b = [Win95,Blue];`<br>`var styles c;`<br><br>`c = a ^ b;`<br>`// c = [Aqua,Blue]` |
| ^ | set ^ set | Exclusive 'or' of two sets. The result is a new set, which contains the enumerators enclosed exclusively either in the left or in the right operand, but not in both:<br><br>`var Unit::Align a = Unit::Align[Top,Left];`<br>`var Unit::Align b = Unit::Align[Top];`<br>`var Unit::Align c;`<br><br>`c = a & b; // c = Unit::Align[ Left ]` |
| && | rect && rect | Rectangle intersection. The result is a rectangle that represents the intersecting area of the both `rect` operands. If the intersecting area is empty, the operator returns an empty rectangle.<br><br>Unlike the rectangle intersection operator `&` empty rectangles are ignored. If one of the operands is an empty rectangle, **no** intersection is build and the operator returns the other operand immediately.<br><br>`var rect a = <10,5,110,220>;`<br>`var rect b = <0,0,0,0>;`<br>`var rect c;`<br><br>`c = a && b;`<br>`// c = <10,5,110,220>` |
| && | bool && bool | Logical 'and'. The result is true if both operands are true.<br><br>`var bool a = true;`<br>`var bool b = true;`<br>`var bool c;`<br><br>`c = a && b; // c = true` |
| \|\| | bool \|\| bool | Logical 'or'. The result is true if one of the operands is true.<br><br>`var bool a = true;` |

| | | |
|---|---|---|
| | | `var bool b = false;`<br>`var bool c;`<br><br>`c = a \|\| b; // c = true` |
| == | `rect == point`<br>`point == rect` | Is point in rectangle test. The result of this expression is true if the given point does lie within the area of the `rect` operand.<br><br>`var rect  a = <0,0,100,200>;`<br>`var point b = <10,20>;`<br>`var point c = <200,300>;`<br>`var bool  d;`<br><br>`d = a == b; // d = true`<br>`d = a == c; // d = false` |
| == | `Op1 == Op2`<br>(any datype) | Equal to. The result of `Op1 == Op2` is true, if both operands are identical. This operator can be used for all data types. The data types of both operands must match so that the comparison can be done.<br><br>`var string a = "Hello";`<br>`var string b = "World";`<br>`var bool c;`<br><br>`c = a == b; // c = false` |
| != | `Op1 != Op2`<br>(any datype) | Not equal to. The result of `Op1 != Op2` is true, if both operands are different. This operator can be used for all data types. The data types of both operands must match so that the comparison can be done.<br><br>`var string a = "Hello";`<br>`var string b = "World";`<br>`var bool c;`<br><br>`c = a != b; // c = true` |
| != | `rect != point`<br>`point != rect` | Is point in rectangle test. The result of this expression is true if the given point does not lie within the area of the `rect` operand.<br><br>`var rect  a = <0,0,100,200>;`<br>`var point b = <10,20>;`<br>`var point c = <200,300>;`<br>`var bool  d;`<br><br>`d = a == b; // d = false`<br>`d = a == c; // d = true` |
| < | `?int?? < ?int??`<br>`float < float`<br>`char < char` | Less than. The result is true, if the left operand is less than the value of the right operand. Both operands must be arithmetic. |

| | | |
|---|---|---|
| | | `var int32 a = 1369;`<br>`var int32 b = 1251;`<br>`var bool c;`<br><br>`c = a < b; // c = false` |
| < | `string < string` | Less than. The result is true, if the left `string` operand is lexicographically less than the value of the right `string` operand.<br><br>`var string a = "cold";`<br>`var string b = "cool";`<br>`var bool c;`<br><br>`c = a < b; // c = true` |
| > | `?int?? > ?int??`<br>`float > float`<br>`char > char` | Greater than. The result is true, if the left operand is greater than the value of the right operand. Both operands must be arithmetic.<br><br>`var int32 a = 1369;`<br>`var int32 b = 1251;`<br>`var bool c;`<br><br>`c = a > b; // c = true` |
| > | `string > string` | Greater than. The result is true, if the left `string` operand is lexicographically greater than the value of the right `string` operand.<br><br>`var string a = "cold";`<br>`var string b = "cool";`<br>`var bool c;`<br><br>`c = a > b; // c = true` |
| <= | `?int?? <= ?int??`<br>`float <= float`<br>`char <= char` | Less than or equal to. The result is true, if the left operand is less than or equal to the value of the right operand. Both operands must be arithmetic.<br><br>`var int32 a = 1369;`<br>`var int32 b = 1369;`<br>`var bool c;`<br><br>`c = a <= b; // c = true` |
| <= | `string <= string` | Less than or equal to. The result is true, if the left `string` operand is lexicographically less than or equal to the value of the right `string` operand.<br><br>`var string a = "cold";`<br>`var string b = "cold";`<br>`var bool c;`<br><br>`c = a <= b; // c = true` |

| | | |
|---|---|---|
| >= | ?int?? >= ?int??<br>float >= float<br>char >= char | Greater than or equal to. The result is true, if the left operand is greater than or equal to the value of the right operand. Both operands must be arithmetic.<br>`var int32 a = 1369;`<br>`var int32 b = 1369;`<br>`var bool c;`<br>`c = a >= b; // c = true` |
| >= | string >= string | Greater than or equal to. The result is true, if the left `string` operand is lexicographically greater than or equal to the value of the right `string` operand.<br>`var string a = "cold";`<br>`var string b = "cold";`<br>`var bool c;`<br>`c = a >= b; // c = true` |

**Table 7–2**

The operators may be used to join an arbitrary number of operands in an expression. The Chora compiler does not define a limit for this. Please note that the data types of the operands used must be suited to the operators. Thus, the following operation is not valid and will result in an error message from the Chora compiler:

```
var rect  Rect1    = <0,0,200,100>;
var rect  Result;

Result = Rect1 + "Hello, world";
```

This example contains at least one error. The '+' operator has been used with a `rect` and a `string` operand. The Chora programming language does not, however, define a '+' operator with which a `rect` and a `string` operands may be combined. As a result, this expression causes a Chora compiler error message.

The expressions are generally evaluated from left to right, whereby partial expressions in parentheses are evaluated first, so that the result of the partial expression can be used in the overall expression as operand.

If the parentheses are missing, the order of evaluation within a complex expression is determined by the precedence of operators. The following priorities exist for the evaluation of complex expressions:

| Priority | Operator name |
|---|---|
| highest | unary + – ~ |
| | * / % |
| | + – |
| | << >> |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | \| |
| | && |
| lowest | \|\| |

**Table 7–3**

This means that in an expression, all unary operators are evaluated first followed by the `*`, `/` and `%` operators. Partial expressions that use a logic 'or' operator `||` are, on the other hand, always evaluated last. This order can be changed explicitly by the programmer through the use of parentheses. As mentioned above, an expression in parentheses is always evaluated first:

```
var int32 Result1;
var int32 Result2;

Result1 = 123 + 456 * 789;
Result2 = ( 123 + 456 ) * 789;
```

This example demonstrates how the order of evaluation of an expression is influenced by the precedence of operators. The first expression without parentheses contains two operators of differing priority: '`*`' and '`+`'. In this case, the '`*`' operator is evaluated first on the basis of its higher priority, and the result of this partial expression is added to `123`. In the second expression, the parentheses force the partial expression with the '`+`' operator to be evaluated before the '`*`' operator. After the operation, `Result1` has a value of `359907`. `Result2`, on the other hand, is initialized with a value of `456831`.

## 7.3 Instant cast operators

Instant cast operators are used whenever an operand in an expression must be subjected to an explicit conversion of its data type. Such a type conversion can be necessary when the data type of the operand can't be used directly in the expression, or when ambiguities exist in the possible evaluation of the expression. When an instant cast operator is used, the desired target data type must be given in parentheses in front of the operand:

```
var float  f = 1.1369;
var int32  i = 4;
var float  fr;
var int32  ir;

fr = f + (float)i;    // fr = 5.1369
ir = (int32)f + i;    // ir = 5
```

The addition of a `float` and an `int32` operand, shown in the above example, would not be possible without an instant cast operator. In contrast to C/C++, the Chora programming language does not allow the addition of two operands of data type `float` and `int32`. In such a situation, the Chora compiler can't conclusively recognize which results should arise from the expression, `float` or `int32`.

This conflict can be resolved easily by subjecting one of the two operands to an explicit type conversion in such a way that the result reflects the programmer's intention. If the programmer expects that the addition yields a `float` value, then the `int32` operand in the expression must be converted into `float`. If, on the other hand, it is expected that the expression yields an `int32` value, then the `float` operand must be converted to `int32`. The result is different in each case: `5.1369` and `5`.

Not every type conversion is allowed. Many of the data types integrated into Chora language are so different that a type conversion applied between the data types would not make much sense. Thus, for example, it is possible to transform an `int32` number into a `string`, whereas a conversion from `color` into `string` is not allowed:

```
var string text;
var int32  volume;
var color  bgColor;
...
text = "Volume level " + (string)volume + " %"; // Ok
text = "The color is " + (string)bgColor;        // Error
```

The following illustration shows the type conversions that are possible among the data types. No other Chora instant data types can be converted:

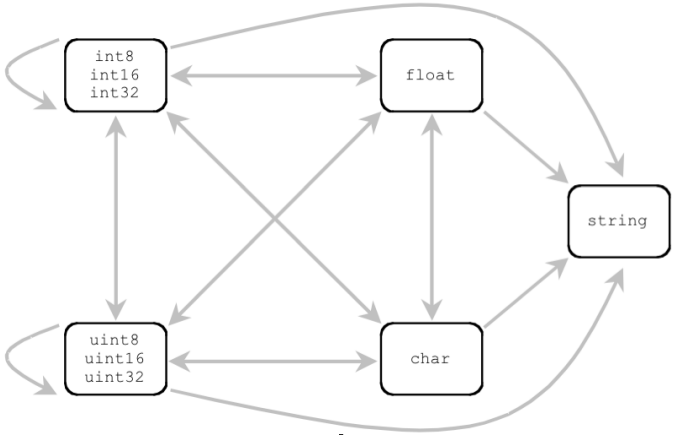



**Figure 7–1**

Aside from explicit type conversion, it is possible to subject the operands in an expression to an automatic, implicit type conversion without having to use the instant cast operator.

For example, the Chora compiler knows that when a value is assigned to a variable, the data type of the assigned value and the data type of the variable in which the value is to be stored must match. If this is not the case, Chora will attempt to perform an appropriate implicit type conversion in order to match the assigned value to the variable's data type. This process is analogous to use of the instant cast operator:

```
var string text;
var int32  volume = 54;
...
text = volume;  // text = "54"
```

The same procedure can be used when a method is called. Here, the method's definition is familiar to the Chora compiler, so that the arguments for the method call can also be subjected to an implicit type conversion.

In a type conversion, the value of the operand is converted to the desired target data type. This conversion can in some cases mean the loss of data, precision or sign. Therefore, the Chora compiler issues a warning if an implicit (automatic) type conversion at runtime could have a possible side effects. The following example demonstrates how the implicit type conversion of `int32` into `int8` leads to a loss of data, since `int8` may store only `8` bits of data and can't accommodate the value `-300`:

```
var int32  i32 = -300;
var int8   i8;
var uint32 u32;
...
i8  = i32;    // Warning data loss: i8  = -44
u32 = i32;    // Warning sign loss: u32 = 0xFFFFFED4

i32 = i8;     // Ok
```

## 7.4 Object cast operators

The object cast operator is a special variant of the type conversion, in which an object's class is dynamically verified at runtime and can be changed. The object type conversion is limited only to the evaluation of the class and the inheritance hierarchy of the affected object. The object itself will not be changed. When an object cast operator is used, the desired name of the target class must be given in parentheses before the object:

```
var Core::View theView = ...;
var Sample::Item item;
...
item = (Sample::Item)theView;
```

Once used, the object cast operator returns the unchanged object if the object is derived from the target class given. In the case that the object is not derived from the target class, `null` (null object) is returned.

An object is derived from a class only if this class appears in the object's inheritance hierarchy. The following illustration gives an example of a simple inheritance hierarchy consisting of `4` classes. The class `Core::View` is the original base class. Starting from this original base class, `Core::Group` was derived. `Core::Group`, in turn, serves as the base class for classes `Sample::Item` and `Sample::Menu`.


Copyright TARA Systems GmbH
www.embedded-wizard.de

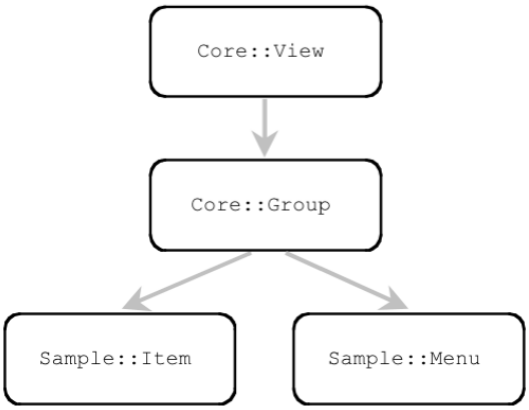



**Figure 7–2**

An object of class `Sample::Item`, therefore, derives from the classes `Core::View`, `Core::Group` and `Sample::Item`. The class `Sample::Menu`, on the other hand, exists outside this object's inheritance hierarchy, so that the object cast operator fails in the conversion from `Sample::Item` to `Sample::Menu` and returns `null`:

```
var object         item = new Sample::Item;
var Core::View     theView;
var Core::Group    theGroup;
var Sample::Item   theItem;
var Sample::Menu   theMenu;
...
theView  = (Core::View)item;     // Ok, theView  = item
theGroup = (Core::Group)item;    // Ok, theGroup = item
theItem  = (Sample::Item)item;   // Ok, theItem  = item
theMenu  = (Sample::Menu)item;   // theMenu  = null
```

Because of Chora's polymorphic character, it is possible at runtime for one and the same variable to point to objects of different classes. The object cast operator is then superbly suited to evaluating the inheritance hierarchy of such an 'unknown' object. In this way, it can be determined at runtime whether the 'unknown' object is indeed derived from a given class, before the object is accessed:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Core::View  theView;

// Sometimes theView points to a Sample::Item object and
// sometimes it points to Sample::Menu object:
if ( someCondition )
  theView = new Sample::Item;
else
  theView = new Sample::Menu;

...

// Hmm, we are not sure. Is 'theView' a Sample::Item
// object?
if ((Sample::Item)theView != null )
{
  // Yes! So it is safe to access this Sample::Item
  // object without any problems ...
  ((Sample::Item)theView).ItemCaption = "Settings";
}

// Or, is 'theView' a Sample::Menu object?
if ((Sample::Menu)theView != null )
{
  // If true, it is safe to access this Sample::Menu
  // object ...
  ((Sample::Menu)theView).ShowMenu();
}
```

Use of the object cast operator in an `if` condition thus ensures that access to the object will occur without a runtime error. Without the `if` statement, the danger exists that a method or property in the context of a `null` object will be used, which invariably leads to a runtime error.

In addition, the object cast operator may also be used with a `null` object. In this case, the object cast operator always returns `null`:

```
var Core::View  theView = null;

if ((Sample::Item)theView != null )
  // This condition will fail ...

if ((Core::View)theView != null )
  // And this condition will fail too ...
```

When using the object cast operator, it should be noted that this type conversion must occur dynamically at runtime. With frequent use, this can lead to a loss of performance. We recommend using the object cast operator only in situations where the class of an object can't always be predicted at runtime. In all other cases, the Chora compiler takes care of statically verifying the variables, classes, methods, etc. in use:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var Sample::Item item = new Sample::Item;

// No object cast necessary, because 'item' is already
// Sample::Item ...
item.ItemCaption = "Setting";
```

The object cast operator functions in a way similar to the `__dynamic_cast<>()` operator familiar from the C++ programming language.

## 7.5 'enum' and 'set' cast operators

The `enum` or `set` cast operator is a special variant of the type conversion, by which an `enum` or a `set` operand is converted into a number operand or vice versa. The following example demonstrates the conversion of an `enum` operand into an `int32` number operand:

```
var Graphics::AlignH align;
var int32            number;

align  = Graphics::AlignH.Center;
number = (int32)align;
```

The resulting value of the conversion depends on the appropriate `enum` definition. If not explicitly specified, the items of an `enum` definition are enumerated automatically starting with the value `0` for the first item, `1` for the second and so far. Assumed that `Center` is the second item within the `Graphics::AlignH` definition, the conversion would return the value `1`. For more details see "Enumerations" (chapter 4.6).

The next kind of possible conversion works in the reverse direction. With this cast operator it is possible to convert a number into a specified `enum` operand:

```
var Graphics::AlignH align;
var int32            number;

number = 1;
align  = (Graphics::AlignH)number;
```

This cast operator evaluates the spefified `enum` definition and verifies, whether there is an item, which does correspond to the given number. If yes, the operator returns the affected item → in our example `Graphics::AlignH.Center` is returned. If the `enum` definition does not contain any corresponding item, the cast operator failes and a runtime error is reported.

In the same manner as described above with the `enum` definitions Chora also provides cast operators for the conversion of `set` operands into numbers and vice versa. The following example demonstrates the conversion from a `set` to a `uint32` operand:

```
var Sample::States states;
var uint32         number;

states = Sample::States[ Visible, Highlighted ];
number = (uint32)states;
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

The resulting value of the conversion depends on the appropriate `set` definition. If not explicitly specified, the items of a `set` definition are enumerated automatically starting with the value `1` for the first item, `2` for the second, `4` for the third and so far. Assumed that within the `Sample::States` definitions `Visible` is the first item and `Highlighted` is the third item, the operator will result the value `1 + 4 = 5`. For more details see "Sets" (chapter 4.7).

The following example demonstrates the conversion in the reverse direction. In this case a number is converted into a `set` operand of the given type:

```
var Sample::States states;
var uint32         number;

number = 5;
states = (Sample::States)number;
```

The idea of this cast operator is the same as the one of the number to `enum` type cast. The operator evaluates at the runtime specified `set` definition and looks for items which will correspond to the given number. If successful the operator returns the affected items as a set. If the `set` definition does not contain any corresponding items, the cast operator fails and a runtime error is reported.

Please note, the verification while the number to `enum` or number to `set` conversion is limited to the prototyping environment only. At the runtime in the target system, the `enum` and `set` operands are already handled as pure numbers → no type conversion and no verification are applied. In this case the resulting values may become invalid. Therefore we recommend always to test the code within the prototyping environment.

## 7.6 'new' operator

The `new` operator is used whenever a new object (an instance) of a particular class must be created at runtime of an GUI application. The name of this desired class must be given to the right of the `new` operator:

```
var Core::Group  theGroup = ...
var Views::Text  theText;

// Create a new instance of the Views::Text class ...
theText = new Views::Text;

// Prepare this new object. Set the font and text
// the object should display on the screen
theText.Bounds = <0,0,100,32>;
theText.Font   = Sample::VerdanaFont;
theText.String = "Hello World";

// And display this object
theGroup.Add( theText, 0 );
```

When executed, the `new` operator causes the Runtime Environment to reserve an area of memory for the new object and to initialize this object.


Copyright TARA Systems GmbH
www.embedded-wizard.de

The initialization process is done according to the definition of the object's class. This definition determines precisely which variables, properties or arrays are initialized, and with what values. The Runtime Environment, therefore, has only to assign initial values to all the relevant variables, properties and arrays. For further details see "Classes" (chapter 4.1). When an object is initialized, the following must be noted:

- The initialization process always starts with the object's oldest base class. This ensures that all variables, properties and arrays defined in a base class have been properly initialized before the initialization of the derived class may proceed.
- In initialization, the Runtime Environment only considers variables, properties and arrays for which an initial value was given in the definition of the class. All other variables, properties and arrays that have not been explicitly initialized still keep a value of `0` (zero) or the value already derived from the base class.
- The initialization order is predetermined by the Z order. The Z order reflects the order in which variables, properties and arrays are listed within the definition of a class. It is possible to alter the initialization order by moving individual variables, properties or arrays up or down within the definition of the class. See control directive `$reorder` (chapter 8.3).
- If the definition of the class contains embedded objects, these are also initialized fully according to the Z order.
- Variables, properties and arrays that are not inherited from the base class are always initialized by a simple assignment of values. The initialization value is copied directly into the memory area of the affected variable, property or array. This means that a property's onset method is never called when the property is initialized the first time.
- The re-initialization of inherited variables and arrays also occurs through a simple assignment of values. The initialization value is copied directly into the memory area of the affected variable or array.
- When re-initializing inherited properties, on the other hand, the onset method of the affected property is called with the initialization value as its argument.
- Once all variables, properties, arrays and embedded objects have been correctly initialized, the user-defined init constructor is called. Here, the programmer can define additional initialization steps that go beyond simple assignment of values. Further details are found in the chapter "Init constructors" (chapter 4.1.5.5).

Thanks to the Garbage Collector, the programmer need not be concerned with destroying unused objects. The Garbage Collector recognizes which objects are no longer in use and automatically disposes of them. Please refer to the chapter "Garbage collection" (chapter 11).

Additionally, if instead of the class name, a `class` expression is placed right to the `new` operator, the expression is evaluated at the runtime and an object of the determined class is created:

```
var class  theClass  = Views::Text;
…
var object theObject = new theClass;
```

If the `new` operator is applied on a `null` class, no object is created and the operator returns `null`.


Copyright TARA Systems GmbH
www.embedded-wizard.de

## 7.7 Instant constructors

Instant constructors take the name of the corresponding data types and expect additional arguments that are dependent on the declaration of the constructor. Instant constructors are a fixed part of the Chora programming language and, as such, can't be changed.

Use of an instant constructor is similar to the call of an ordinary method. The instant constructor expects arguments that it then uses to compute a value. The value thus computed is then returned and can be used in an expression. For example: the instant constructor `rect()` takes the `4` delivered arguments `x`, `y`, `x + 100`, `y + 100` and builds a `rect` value, which it returns, so that this value can be assigned to `bounds`:

```
var rect bounds;

bounds = rect( x, y, x + 100, y + 200 );
```

One could compare an instant constructor to a very powerful type conversion operator that combines multiple arguments and that can convert them to the desired data type.

The Chora programming language supports the following instant constructors:

| Instant constructor declaration | Description |
|---|---|
| **string string**<br>(<br>  arg int32 **aValue,**<br>  arg int32 **aNoOfDigits**<br>) | This constructor converts the given signed number `aValue` in a `string` and returns the result.<br>The constructor adds leading zeros '`0`' until the resulted string has reached the length given in the `aNoOfDigits` argument.<br>If the number is negative, the constructor adds '–' minus sign.<br>`var string text;`<br>`var int32 pegel = -12;`<br>`// text = "-0012"`<br>`text = string( pegel, 5 );` |
| **string string**<br>(<br>  arg uint32 **aValue,**<br>  arg int32 **aNoOfDigits**<br>) | This instant constructor converts the given unsigned number `aValue` in a `string` and returns the result.<br>The constructor adds leading zeros '`0`' until the resulted string has reached the length given in the `aNoOfDigits` argument.<br>`var string text;`<br>`var uint32 volume = 55;`<br>`// text = "055"` |

| | |
|---|---|
| | `text = string( volume, 3 );` |
| **string string**<br>(<br>  arg float **aValue,**<br>  arg int32 **aNoOfDigits,**<br>  arg int32 **aPrecision**<br>) | This constructor converts the given floating point number `aValue` in a `string` and returns the result.<br>The constructor adds leading zeros '`0`' until the resulted string has reached the length given in the `aNoOfDigits` argument.<br>The argument `aPrecision` defines the number of digits after the point.<br>If the `aValue` number is negative, the constructor adds '`-`' minus sign.<br>`var string text;`<br>`var float freq = 96.5349;`<br>`// text = "96.53"`<br>`text = string( freq, 0, 2 );` |
| **string string**<br>(<br>  arg char **aChar,**<br>  arg int32 **aCount**<br>) | This constructor creates a string with the given length `aCount` and fills the whole string with the copy of the character `aChar`.<br>`var string text;`<br>`// text = "****"`<br>`text = string( '*', 4 );` |
| **string string**<br>(<br>) | This constructor does not expect an argument. It returns merely an empty string.<br>`var string text;`<br>`// text = ""`<br>`text = string();` |
| **color color**<br>(<br>  arg uint8 **aRed,**<br>  arg uint8 **aGreen,**<br>  arg uint8 **aBlue**<br>) | This constructor initializes a new `color` with the given values for red, green and blue color components and returns the initialized color.<br>The alpha component is initialized with `255`.<br>`var color bgColor;`<br>`var uint8 r = 0x12;`<br>`var uint8 g = 0x34;`<br>`var uint8 b = 0x56;`<br>`// bgColor = #123456FF`<br>`bgColor = color( r, g, b );` |
| **color color**<br>(<br>  arg uint8 **aRed,** | This constructor initializes a new `color` with the given values for red, green, blue and alpha color components and returns the initialized color. |

| | |
|---|---|
| arg uint8 **aGreen**,<br>arg uint8 **aBlue**,<br>arg uint8 **aAlpha**<br>) | var color bgColor;<br>var uint8 r = 0x12;<br>var uint8 g = 0x34;<br>var uint8 b = 0x56;<br>var uint8 a = 0x78;<br><br>// bgColor = #12345678<br>bgColor = **color( r, g, b, a );** |
| **color color**<br>(<br>) | This constructor does not expect an argument. It returns merely a transparent color.<br><br>var color bgColor;<br><br>// bgColor = #00000000<br>bgColor = **color();** |
| **rect rect**<br>(<br>arg int32 **aX1**,<br>arg int32 **aY1**,<br>arg int32 **aX2**,<br>arg int32 **aY2**<br>) | This constructor initializes a new rectangle with the given `aX1`, `aY1`, `aX2`, `aY2` coordinates and returns the initialized `rect`.<br><br>var rect r;<br>var int32 x1 = 10;<br>var int32 y1 = 20;<br>var int32 x2 = 30;<br>var int32 y2 = 40;<br><br>// r = <10,20,30,40><br>r = **rect(** x1, y1, x2, y2 **);** |
| **rect rect**<br>(<br>arg point **aPoint1**,<br>arg point **aPoint2**<br>) | This constructor initializes a new rectangle with the given points `aPoint1` and `aPoint2` and returns the initialized `rect`.<br><br>var rect r;<br>var point p1 = <10,20>;<br>var point p2 = <30,40>;<br><br>// r = <10,20,30,40><br>r = **rect(** p1, p2 **);** |
| **rect rect**<br>(<br>) | This constructor does not expect an argument. It returns merely an empty rectangle.<br><br>var rect r;<br><br>// r = <0,0,0,0><br>r = **rect();** |
| **point point**<br>(<br>arg int32 **aX**,<br>arg int32 **aY**<br>) | This constructor initializes a new point with the given `aX` and `aY` values and returns the initialized `point`.<br><br>var point p;<br>var int32 x = 10; |

| | |
|---|---|
| | `var int32 y = 20;`<br><br>`// p = <10,20>`<br>`p = point( x, y );` |
| **`point point`**<br>`(`<br>`)` | This constructor does not expect an argument. It returns merely a null point.<br><br>`var point p;`<br><br>`// p = <0,0>`<br>`p = point();` |

**Table 7–4**

## 7.8 Instant methods

Instant methods simplify access to the instant data types. The programmer can use the instant methods to execute an operation on an instant data type. He can, for example, search a string for a particular character using the instant method `find()`:

```
var string text = "Hello world!";
var int32  inx;

// Search for the first occurrence of the character 'w'
inx = text.find( 'w' );  // inx = 6
```

From the programmer's point of view, instant methods are used like the methods of an object with the distinction that instant methods are always used within the context of an instant data type.

Instant methods are a fixed part of the Chora programming language and can therefore not be changed.

The Chora programming language supports the following instant methods:

| Instant method declaration | Description |
|---|---|
| `string` **`string.left`**<br>`(`<br>`  arg int32` **`aCount`**<br>`)` | The `left()` method extracts the first (that is, leftmost) `aCount` characters from the string `aString` and returns a copy of the extracted substring.<br><br>If `aCount` exceeds the string length, then the entire string is returned.<br><br>If `aCount == 0` the method returns an empty string `""`.<br><br>`var string s1 = "Hello World";`<br>`var string s2;`<br><br>`// s2 = "Hello"` |

| | |
|---|---|
| | `s2 = s1.left( 5 );` |
| `string string.right ( arg int32 aCount )` | The `right()` method extracts the last (that is, rightmost) `aCount` characters from the string `aString` and returns a copy of the extracted substring.<br>If `aCount` exceeds the string length, then the entire string is returned.<br>If `aCount == 0` the method returns an empty string "".<br>`var string s1 = "Hello World";`<br>`var string s2;`<br>`// s2 = "World"`<br>`s2 = s1.right( 5 );` |
| `string string.middle ( arg int32 aIndex, arg int32 aCount )` | The `middle()` method extracts a substring of length `aCount` characters from the string, starting at position `aIndex` within this string. The method returns a copy of the extracted substring.<br>The argument `aIndex` contains the index of the first character in the string where the extracted substring begins. The characters in a string are counted from `0` (zero).<br>If `aCount == 0` the method returns an empty string "".<br>`var string s1 = "Hello World";`<br>`var string s2;`<br>`// s2 = "llo"`<br>`s2 = s1.middle( 2, 3 );` |
| `string string.insert ( arg string aString, arg int32 aIndex )` | The `insert()` method inserts the substring `aString` at the given position `aIndex` within the string.<br>If `aIndex` is `<= 0` (zero), the insertion will occur before the entire string.<br>If `aIndex` is greater than the length of the string, the function will concatenate `aString` at the end of the string.<br>`var string s1 = "Hello World";`<br>`var string s2;`<br>`// s2 = "Hello nice World"`<br>`s2 = s1.insert( "nice ", 6 );` |
| `string string.remove` | The `remove()` method removes up to `aCount` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| (<br>  arg int32  **aIndex**,<br>  arg int32  **aCount**<br>) | characters from the string starting with the character at position aIndex and returns a copy of the resulted string.<br><br>`var string s1 = "Hello World";`<br>`var string s2;`<br><br>`// s2 = "Hello"`<br>`s2 = s1.remove( 5, 6 );` |
| int32 **string.find**<br>(<br>  arg char  **aChar**,<br>  arg int32  **aStartIndex**<br>) | This find() method searches the string for the first occurrence of a character aChar.<br><br>find() starts the search operation at the position aStartIndex in the string. The function returns the index of the first character in the string that matches the requested character aChar or -1 if aChar has not been found in the string.<br><br>`var string s1 = "Hello World";`<br>`var int32  inx;`<br><br>`// inx = 7 --> second 'o'`<br>`inx = s1.find( 'o', 5 );` |
| int32 **string.find**<br>(<br>  arg string **aString**,<br>  arg int32  **aStartIndex**<br>) | This second find() method searches the string for the first occurrence of a substring aString.<br><br>find() starts the search operation at the position aStartIndex in the string. The function returns the index of the first character in the string that matches the requested substring aString or -1 if aString has not been found in the string.<br><br>`var string s1 = "Hello World";`<br>`var int32  inx;`<br><br>`// inx = 7`<br>`inx = s1.find( "orl", 0 );` |
| bool **styles.contains**<br>(<br>  arg styles **aStyles**<br>) | This contains() method determines, whether the given styles set aStyles is completely enclosed in another styles set.<br><br>If the styles set is enclosed, the method returns true. Otherwise false is returned.<br><br>`var styles a = [Aqua,Win95];`<br>`var styles b = [Win95];`<br>`var bool   c;`<br><br>`// c = true`<br>`c = a.contains( b );` |
| bool **set.contains** | This contains() method determines, whether |

| | |
|---|---|
| (<br>  arg set-type **aSet**<br>) | the given set `aSet` is completely enclosed in another set.<br>If the set is enclosed, the method returns `true`. Otherwise `false` is returned.<br>`var Unit::Align a = Unit::Align[Top,Left];`<br>`var Unit::Align b = Unit::Align[Left];`<br>`var bool c;`<br>`// c = true`<br>`c = a.contains( b );` |

**Table 7–5**

## 7.9 Instant properties

Instant properties simplify access to the instant data types. In the case of complex instant data types, such as `rect`, `point` or `color`, instant properties allow direct access to the elements of the data type. For example, the value of the alpha component in a color data type can be queried or changed using the instant property `alpha`.

```
var color  bgColor = #FFFF0078;
var int32  a;

// Query the alpha value of the bgColor
a = bgColor.alpha;  // a = 0x78
```

From the programmer's point of view, instant properties are used like the properties of an object with the distinction that instant properties are always used within the context of an instant data type.

Instant properties are a fixed part of the Chora programming language and can therefore not be changed.

The Chora programming language supports the following instant properties:

| Instant property declaration | Description |
|---|---|
| `char` **`char.upper`** | The read access to the instant property `upper` of a character returns its uppercase copy. The original character will be unchanged.<br>`var char a = 'a';`<br>`var char b = 'A';`<br>`var char c;`<br>`c = a.upper;  // c = 'A'`<br>`c = b.upper;  // c = 'A'`<br>This instant property is read only. You can't assign any values to `upper`. |
| `char` **`char.lower`** | The read access to the instant property `lower` of a character returns its lowercase copy. The original |

| | |
|---|---|
| | character will be unchanged.<br>`var char a = 'a';`<br>`var char b = 'A';`<br>`var char c;`<br>`c = a.lower;  // c = 'a'`<br>`c = b.lower;  // c = 'a'`<br>This instant property is read only. You can't assign any values to `lower`. |
| `string` **`string.upper`** | The read access to the instant property `upper` of a string returns its uppercase copy. The original string will be unchanged.<br>`var string a = "Hello";`<br>`var string c;`<br>`c = a.upper;  // c = "HELLO"`<br>This instant property is read only. You can't assign any values to `upper`. |
| `string` **`string.lower`** | The read access to the instant property `lower` of a string returns its lowercase copy. The original string will be unchanged.<br>`var string a = "HEllo";`<br>`var string c;`<br>`c = a.lower;  // c = "hello"`<br>This instant property is read only. You can't assign any values to `lower`. |
| `int32` **`string.length`** | The read access to the instant property `length` of a string returns the number of characters contained within this string.<br>If the string is empty, the property returns `0`.<br>`var string a = "Hello World!";`<br>`var string b = "";`<br>`var int32  c;`<br>`c = a.length;  // c = 12`<br>`c = b.length;  // c = 0`<br>This instant property is read only. You can't assign any values to `length`. |
| `uint8` **`color.red`** | The instant property `red` corresponds to the red value of the color. The read access to this property returns this red value and the write access changes it. The other values: green, blue and alpha are not affected. |


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var color a = #123456FF;
var uint8 c;

// Get the red value
c = a.red;     // c = 0x12

// change the red value
a.red = 0xEF; // a = #EF3456FF
```

uint8 **color.green**

The instant property `green` corresponds to the green value of the color. The read access to this property returns this green value and the write access changes it. The other values: red, blue and alpha are not affected.

```
var color a = #123456FF;
var uint8 c;

// Get the green value
c = a.green;     // c = 0x34

// change the green value
a.green = 0xEF; // a = #12EF56FF
```

uint8 **color.blue**

The instant property `blue` corresponds to the blue value of the color. The read access to this property returns this blue value and the write access changes it. The other values: red, green and alpha are not affected.

```
var color a = #123456FF;
var uint8 c;

// Get the blue value
c = a.blue;     // c = 0x56

// change the blue value
a.blue = 0xEF; // a = #1234EFFF
```

uint8 **color.alpha**

The instant property `alpha` corresponds to the alpha value of the color. The read access to this property returns this alpha value and the write access changes it. The other values: red, green and blue are not affected.

```
var color a = #123456FF;
var uint8 c;

// Get the alpha value
```

| | |
|---|---|
| | ```<br>c = a.alpha;    // c = 0xFF<br><br>// change the alpha value<br>a.alpha = 0x00; // a = #12345600<br>``` |
| int32 **rect.x1** | The instant property `x1` corresponds to the X1 (left) coordinate of the rectangle. The read access to this property returns this coordinate value and the write access changes it. The other coordinate values are not affected.<br>```<br>var rect  a = <10,20,30,40>;<br>var int32 c;<br><br>// Get the x1 coordinate<br>c = a.x1;    // c = 10<br><br>// change the x1 coordinate<br>a.x1 = 0; // a = <0,20,30,40><br>``` |
| int32 **rect.y1** | The instant property `y1` corresponds to the Y1 (top) coordinate of the rectangle. The read access to this property returns this coordinate value and the write access changes it. The other coordinate values are not affected.<br>```<br>var rect  a = <10,20,30,40>;<br>var int32 c;<br><br>// Get the y1 coordinate<br>c = a.y1;    // c = 20<br><br>// change the y1 coordinate<br>a.y1 = 0; // a = <10,0,30,40><br>``` |
| int32 **rect.x2** | The instant property `x2` corresponds to the X2 (right) coordinate of the rectangle. The read access to this property returns this coordinate value and the write access changes it. The other coordinate values are not affected.<br>```<br>var rect  a = <10,20,30,40>;<br>var int32 c;<br><br>// Get the x2 coordinate<br>c = a.x2;    // c = 30<br><br>// change the x2 coordinate<br>a.x2 = 100; // a = <10,20,100,40><br>``` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| int32 **rect.y2** | The instant property y2 corresponds to the Y2 (bottom) coordinate of the rectangle. The read access to this property returns this coordinate value and the write access changes it. The other coordinate values are not affected.<br>`var rect a = <10,20,30,40>;`<br>`var int32 c;`<br>`// Get the y2 coordinate`<br>`c = a.y2; // c = 40`<br>`// change the y2 coordinate`<br>`a.y2 = 100; // a = <10,20,30,100>` |
| int32 **rect.w** | The instant property w corresponds to the width of the rectangle. The read access to this property returns the width and the write access changes it.<br>The width of the rectangle is calculated by subtracting the coordinates of the left border X1 from the right border X2.<br>`var rect a = <10,20,30,50>;`<br>`var int32 c;`<br>`// Get the width`<br>`c = a.w; // c = 20`<br>`// change the width`<br>`a.w = 100; // a = <10,20,110,50>` |
| int32 **rect.h** | The instant property h corresponds to the height of the rectangle. The read access to this property returns the height and the write access changes it.<br>The height of the rectangle is calculated by subtracting the coordinates of the top border Y1 from the bottom border Y2.<br>`var rect a = <10,20,30,50>;`<br>`var int32 c;`<br>`// Get the height`<br>`c = a.h; // c = 30`<br>`// change the height`<br>`a.h = 100; // a = <10,20,30,120>` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| point **rect.point1** | The instant property point1 corresponds to the top-left edge of the rectangle. The read access to this property returns the coordinates of this edge and the write access changes them. The other edge is not affected.<br>`var rect a = <10,20,30,50>;`<br>`var point c;`<br>`// Get the top-left edge`<br>`c = a.point1; // c = <10,20>`<br>`// Move the top-left edge`<br>`a.point1 = <5,7>; //a=<5,7,30,50>` |
| point **rect.point2** | The instant property point2 corresponds to the bottom-right edge of the rectangle. The read access to this property returns the coordinates of this edge and the write access changes them. The other edge is not affected.<br>`var rect a = <10,20,30,50>;`<br>`var point c;`<br>`// Get the bottom-right edge`<br>`c = a.point2; // c = <30,50>`<br>`// Move the bottom-right edge`<br>`a.point2 = <35,57>;`<br>`// a = <10,20,35,57>` |
| point **rect.origin** | The instant property origin corresponds to the origin of the rectangle. The read access to this property returns the coordinates of this origin edge and the write access moves the whole rectangle, so the size of the rectangle doesn't change.<br>This behavior is the most important difference to the point1 instant property, which changes the coordinates of the top-left edge only.<br>`var rect a = <10,20,30,50>;`<br>`var point c;`<br>`// Get the origin edge`<br>`c = a.origin; // c = <10,20>`<br>`// Move the whole rectangle`<br>`a.origin = <5,7>; //a=<5,7,25,37>` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| point **rect.size** | The instant property size corresponds to the size of the rectangle. The read access to this property returns the size (width, height) and the write access changes it. The origin of the rectangle still keep unchanged.<br>The size of the rectangle is calculated by subtracting the coordinates of the top-left edge X1, Y1 from the bottom-right edge X2, Y2.<br>`var rect a = <10,20,30,50>;`<br>`var point c;`<br><br>`// Get the size of the rectangle`<br>`c = a.size; // c = <20,30>`<br><br>`// Resize the rectangle`<br>`a.size = <10,15>;`<br>`// a = <10,20,20,35>` |
| rect **rect.orect** | The read access to the instant property orect of a rectangle returns a copy of this rectangle moved to the zero origin <0,0>. The size of the rectangle doesn't change.<br>`var rect a = <10,20,30,50>;`<br>`var rect c;`<br><br>`c = a.orect; // c = <0,0,20,30>`<br>This instant property is read only. You can't assign any values to orect. |
| int32 **rect.area** | The instant property area can be used to determine the area occupied by the rectangle. The area is calculated by multiplying the width and the height of the rectangle. If the width, or the height are negative, the resulted area can be negative too. If the width or the height are zero, an empty area results.<br>`var rect a = <10,20,30,50>;`<br>`var int32 c;`<br><br>`c = a.area; // c = 600`<br>This instant property is read only. You can't assign any values to area. |
| point **rect.center** | The instant property center calculates the coordinates of the point in center of the rectangle.<br>`var rect a = <10,20,30,50>;`<br>`var point c;` |

| | | |
|---|---|---|
| | | `c = a.center; // c = <20,35>`<br><br>This instant property is read only. You can't assign any values to `center`. |
| `bool` | **`rect.isempty`** | The instant property `isempty` is used to determine whether a rectangle is empty or not. An empty rectangle has an area equal to or less than zero (`x1 >= x2` or `y1 >= y2`).<br><br>`var rect a = <10,20,30,50>;`<br>`var rect b = <10,20,10,50>;`<br>`var bool c;`<br><br>`c = a.isempty; // c = false`<br>`c = b.isempty; // c = true`<br><br>This instant property is read only. You can't assign any values to `isempty`. |
| `int32` | **`point.x`** | The instant property `x` corresponds to the X coordinate of the point. The read access to this property returns this coordinate value and the write access changes it. The other coordinate value Y is not affected.<br><br>`var point a = <10,20>;`<br>`var int32 c;`<br><br>`// Get the x coordinate`<br>`c = a.x; // c = 10`<br><br>`// change the x coordinate`<br>`a.x = 0; // a = <0,20>` |
| `int32` | **`point.y`** | The instant property `y` corresponds to the Y coordinate of the point. The read access to this property returns this coordinate value and the write access changes it. The other coordinate value X is not affected.<br><br>`var point a = <10,20>;`<br>`var int32 c;`<br><br>`// Get the y coordinate`<br>`c = a.y; // c = 20`<br><br>`// change the y coordinate`<br>`a.y = 0; // a = <10,0>` |
| `bool` | **`styles.isempty`** | The instant property `isempty` is used to determine whether a styles set is empty or not. An empty styles set does not enclose any style names: |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| | ```<br>var styles a = [Aqua];<br>var styles b = [];<br>var bool   c;<br><br>c = a.isempty; // c = false<br>c = b.isempty; // c = true<br>```<br>This instant property is read only. You can't assign any values to `isempty`. |
| `bool` **`set.isempty`** | The instant property `isempty` is used to determine whether a set is empty or not. An empty set does not enclose any enumerators:<br>```<br>var Unit::Align a = Unit::Align[Top,Left];<br>var Unit::Align b = [];<br>var bool       c;<br><br>c = a.isempty; // c = false<br>c = b.isempty; // c = true<br>```<br>This instant property is read only. You can't assign any values to `isempty`. |
| `int32` **`array.size`** | The read access to the instant property `size` of an array returns the number of elements within this array → the max. capacity of the array.<br>```<br>array string a[4];<br>array string b[4,2];<br>var int32  c;<br><br>c = a.size;    // c = 4<br>c = b.size;    // c = 8<br>```<br>This instant property is read only. You can't assign any values to `size`. |

**Table 7–6**

When using instant properties, it must be noted that not every instant property allows an assignment. Thus, for example, it is not possible to use an assignment to modify the length of a string. The instant property length is exclusively read-only:

```
var string Text = "Hello World!";
var int32  l;
...
Text.length = 20;   // Error!
l = Text.length;    // Ok
```

Assignment to an instant property within the scope of a constant or a literal is also not allowed, as neither the constant nor the literal can be changed. In this case as well, a Chora compiler error is reported:

```
var point p;

<10,20,30,40>.point2 = <0,0>;    // Error!
p = <10,20,30,40>.point2;        // Read access is Ok
```

## 7.10 Index operator

Using an index operator, it is possible to read a character from a string or to change a character into a string. When a character is to be accessed, the number (the index) of the desired character must be given in brackets:

```
var string Text = "Hello World!";
var char   C;
var int32  inx  = 5;
...
C = Text[0];     // C = 'H'
C = Text[1];     // C = 'e'
C = Text[11];    // C = '!'

Text[ inx + 1 ] = ' ';
...
```

The characters in a string are always counted from `0` (zero). Thus, the first character is number `0`, the second number `1`, and so on. Access to a nonexistent character causes an error at runtime. Thus, it is not possible, as in C/C++ to access characters outside of a string:

```
var string Text = "Hello World!";
var char   C;
...
C = Text[0];     // Ok
C = Text[11];    // Ok
C = Text[12];    // Ok → returns zero terminator '\x0' ,
C = Text[13];    // Syntax or runtime error !!!
C = Text[-1];    // Syntax or runtime error !!!
```

It is recommended that the instant property `length` of a string be evaluated before the index operator is used on the string. The instant property `length` returns the number of characters in a string:

```
var string Text = "Hello World!";
...
if ( Text.length > 0 )
  Text[0] = ...
```

Use of the index operator is similar to an access to a 1-dimensional array. In order to access a character in a string array, the index to the array and the index of the desired character must be given one after the other in brackets:

```
array string Items[ 16 ];
var char     C;
...
Items[ 10 ] = "Hello World!";
...
C = Items[10][0];     // C = 'H'
C = Items[10][1];     // C = 'e'
C = Items[10][11];    // C = '!'
```

The following example demonstrates the use of the index operator in order to convert all characters in all strings from upper to lower case:

```
array string Items[ 16 ];
var int32    i1;
var int32    i2;
...
// All strings in the array Items[] are converted to
// lower-case, character by character ...
for ( i1 = 0; i1 < 16; i1 = i1 + 1 )
  for ( i2 = 0; i2 < Items[ i1 ].length; i2 = i2 + 1 )
    Items[ i1 ][ i2 ] = Items[ i1 ][ i2 ].lower;
```

The index operator can also be used in a complex expression if the expression results in a string and one wish to query a character within this string:

```
var char c;
...
c = ( "Hello " + GetUserName())[ 6 ];
```

Unlike reading a character, modifying a character within a string is possible only with limitations. If the string is a literal, a constant or an expression, the string can't be changed — an assignment is not possible:

```
var char   c;
var string text = "Hello World!";
...
// Assignment
( "Hello " + GetUserName())[ 6 ] = ' '; // Syntax error
"Hello"[ 4 ]                     = '*'; // Syntax error
Sample::TheStringConstant[ 0 ]   = '*'; // Syntax error
text[0]                          = 'h'; // Ok

// Read access
c = ( "Hello " + GetUserName())[ 6 ];   // Ok
c = "Hello"[ 4 ];                       // Ok
c = TheStringConstant[ 0 ];             // Ok
```

Please note, that the Chora `string` index operator should be used very carefully, because it may lead to some side-effects: By using the Chora `string` index operator it is possible to encounter unexpected behavior in the software, if you are modifying a string, which effectively is a string constant. In this case the modification is performed directly in the origin string constant and therefore it affects all parts of the software. For example, the following code does modify the constant string:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var string a = “Hello World”;
var string b = “Hello World”;

a[0] = ‘X’; // a is “Xello World”

trace b;    // b is ALSO “Xello World”
```

In such cases it is better to modify the strings by using the `string` instant methods `remove()` and `insert()`. These methods do not have any side effects. For more details see “Instant methods” (chapter 7.8).

## 7.11 Assignment operator

The Chora assignment operator '=' is used to store the result of an expression in variables, properties or arrays. The result is 'assigned' to the variable:

```
var Sample::Item Item = new Sample::Item;
var string       Caption;
array rect       Items[5];
...
Caption  = "Hello " + GetUserName();
Items[2] = Item.Bounds & Clipping;
Item.Bounds.x2 = Item.Bounds.x2 + 16;
...
```

When an assignment is made to a variable or to an array item, the result of the assigned expression is copied directly into the memory of the variable or the array.

When an assignment is made to a property, the onset method of that property is always invoked with the result of the expression as its hidden `value` argument. The new property value is not copied to the memory at that time. Only in the onset method the programmer can decide what should happen with the assigned value. Details of the onset method may be found in "onset methods" (chapter 4.1.5.3).

The Chora assignment operator subjects the result of the expression to an implicit (automatic) type conversion if the data type of the expression does not match the data type of the variable, property or array:

```
var Core::View Menu
Menu = new Sample::Item;
```

The above assignment therefore uses an implicit type conversion in order to convert the expression from data type `Sample::Item` to `Core::View`. The same result can be achieved by using an explicit typecast operator:

```
var Core::View Menu
Menu = (Core::View)new Sample::Item;
```

The implicit type conversion follows the same rules as the explicit typecast operators described in "Instant cast operators" (chapter 7.3) and "Object cast operators" (chapter 7.4).


Copyright TARA Systems GmbH
www.embedded-wizard.de

## 7.12 'parentthis' operator

The parentthis operator is used whenever an embedded object needs to access its superior object. In this case the operator determines this superior object and returns it. If the object has not been embedded, parentthis will always return null.

This operator is very useful, when an object needs to know, whether it is embedded within a superior object, or whether it has been created dynamically by the new operator:

```
if ( parentthis != null )
  … 'this' object is embedded within parentthis.
else
  … 'this' object has been created by the new operator
```

For more details about the new operator see "'new' operator" (chapter 7.6).

## 7.13 'classof' operator

The classof operator determines the class of an object. The affected object expression must be given to the right of the classof operator:

```
var object anObject = GetAnObject();
var class  theClass = classof anObject;
```

The operator returns a value of type class. If the operator is applied on null object, the operator returns null class. This operator is very useful, when the class of an object should be determined and stored for further processing. For example, the returned class can be passed to the new operator in order to create an object of the given class:

```
var class theClass = classof anObject;
…
var object theObject = new theClass;
```

In order to verify, whether the class is derived from a desired class, the class cast operator may be used. The class cast operator returns null, if the tested class is not derived from the desired class:

```
var class theClass = classof anObject;

if ((Core::Group)theClass != null )
  // Yes, it is derived from Core::Group
```

For more details see "'new' operator" (chapter 7.6) and the description of the class type in "Instant data types" (chapter 6.1).

## 7.14 Build-in functions

Besides the instant operators and methods, Chora also provides some build-in functions useful for common mathematical calculations. Following functions are available:

```
float math_sin( float aAngle );
float math_cos( float aAngle );
float math_asin( float aValue );
float math_acos( float aValue );
float math_pow( float aA, float aB );
float math_sqrt( float aValue );
float math_rand( float aValue1, float aValue2 );
int32 math_rand( int32 aValue1, int32 aValue2 );
```

The trigonometric functions `math_sin()` and `math_cos()` calculate the sine and cosine values of the given angle `aAngle` expressed in degrees. For example:

```
var float angle = 60.0;
var float x     = math_cos( angle ); // x = 0.5
var float y     = math_sin( angle ); // y = 0.86602…
```

The inverse trigonometric functions `math_asin()` and `math_acos()` calculate the arcsine and arccosine of the given value `aValue`. The functions return the angle in degree. For example:

```
var float angle = math_acos( 0.5 ); // angle = 60.0
var float angle = math_asin( 0.5 ); // angle = 30.0
```

The function `math_pow()` calculates value of `aA` power `aB`. For example:

```
var float x  = 2.5;
var float x2 = math_pow( x, 2.0 ); // x = 6.25
var float x3 = math_pow( x, 3.0 ); // x = 15.625
```

The function `math_sqrt()` calculates the square root value of the given `aValue`. For example, it is used for vector length calculation:

```
var float vecX = 10.5
var float vecY = 4.6;
var float len  = math_sqrt( vecX * vecX + vecY * vecY );
                 // len = 11.4634…
```

The both functions `math_rand()` return random values from the given range `aValue1 .. aValue2`. The quality of the returned values depends on the random generator implementation available in the underlying target system. Usually, the typical solution will provide pseudo random numbers only.

Please note the existing limitation in the usage of the `int32` version of `math_rand()`. The function can return incorrect values in case of too large range specified in the `aValue1` and `aValue2` parameters. If using this `int32` version, please avoid ranges larger than `32767`.


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 8 Chora preprocessor

The Chora programming language allows the development of platform-independent GUI applications. In the most favorable case, such an GUI application may be ported directly to a new platform. In most cases, however, minor adjustments must be made to the GUI application in order to fit the resolution and color depth of the new target system. The logic of the native methods, native statements and the inline code definitions must also be adjusted accordingly.

In order to simplify all the necessary changes and adjustments within the GUI application, the Chora programming language contains a preprocessor. This preprocessor analyses the source code of the GUI application and enables conditional code generation that depends on the chosen platform, profile or other user-defined settings:

- The preprocessor's first job is to evaluate and replace macros. The programmer uses macros in order to store arbitrary character strings under a unique name. If the preprocessor finds such a macro name in the GUI application, it will automatically replace that name with the relevant character string. With the aid of a macro, the screen resolution, for example, can be set platform-independently. Wherever the GUI application has to query the screen resolution, the macro can be used. → The code of the GUI application thus remains platform-independent. See "Profiles and macros" (chapter 3.2).
- The second job for the preprocessor is the conditional code generation through the evaluation of the `$if`, `$else`, `$elseif` and `$endif` directives. With the aid of these directives, some code lines of a method or an inline code definition can be excluded from the code generation.
- Finally, the preprocessor is concerned with evaluating the control directives. These directives control the code generator and determine how the GUI application will appear in the Embedded Wizard composer editor. For example, the directive `$output` determines whether the class definition that immediately follows it should be passed on to the code generator, or not. Using `$output false` the class can then be excluded from the code generation.

In general, the Chora preprocessor works similar to the preprocessor familiar from C or C++.

## 8.1 Macro evaluation

In order for the preprocessor to be able to recognize and replace a macro in the code of the GUI application, the name of the macro must be introduced with a '`$`' (dollar sign):

```
class Menu : Core::Group
{
  ...
  inherited property Bounds = <0,0,$Size>;
  ...
}
```

This example demonstrates the use of the macro `Size` in the initialization of the property `Bounds` of a menu. The Chora preprocessor recognizes from the leading '`$`' sign that it is dealing with a macro, and replaces the name of the macro, including the '`$`' sign, with the macro's contents. Assuming the macro contains the character string `640,480`, then `Bounds` is initialized with the rectangle `<0,0,640,480>`.

In order for the preprocessor to be able to find and replace a macro, each macro must already have been defined explicitly. Macros are defined in profiles → Each profile can (or must, even) contain its own set of macro definitions, whereby the content of the macros may differ from profile to profile. Thus, the macro `Size` could be defined in a second profile with the character string `1024,768` if the relevant target system allows a high screen resolution:

```
// Target with a low screen resolution
profile Target1 : Tara.Win32.RGBA8888
{
  macro Size = 640,480;
}

// Target with a high screen resolution
profile Target2 : Tara.Win32.RGBA8888
{
  macro Size = 1024,768;
}
```

To allow the preprocessor to access the macros defined in a profile, the relevant profile must be selected explicitly for the code generation. For example, when the Chora compiler is invoked, the name of the desired profile must be passed on to the compiler as the last parameter:

```
chorac   testproject.ewp   Target2
```

Each time the preprocessor recognizes a macro, it searches the currently selected profile for the corresponding macro definition, gets the character string stored there and use it to replace the found macro occurrence. In the case that the required macro definition could not be found in the currently selected profile, the Chora compiler will report a warning. How profiles and macros are defined is explained in the chapter "Profiles and macros" (chapter 3.2).

A major difference between the Chora preprocessor and the C, C++ preprocessors is the limitation that Chora macros can't be placed anywhere within the program code. Thus, the leading keywords (`class`, `method`, `enum`, `property`, `var`, `inherited`, `native`, ...) can't be replaced using macros. For example, it is not possible to replace the keyword `class` in the definition of a class by a `$CLASS` macro:

```
$CLASS Menu : Core::Group
{
  ...
}
```

Even when the macro `$CLASS` would contain the character string `class`, the above example would cause a Chora compiler error, because the macro is misplaced. In the same way, it is not possible to use macros in order to specify the type of a variable or the base class, an other class is derived from. The following example will also cause a Chora compiler error:

```
class Menu : $MenuSuperClass
{
  var $MyVariableType Counter;
}
```

Such advanced tricks, can be realized much more effectively by using the variants → see "Variants" (chapter 4.8). For example, the class for a menu item can be extended by an additional appearance and behavior – depending on the selected profile or style.

In most cases, however, macros are used only in the initialization of variables, properties, and arrays, or directly in the logic of methods. That already suffices to make the GUI application product- and platform-independent:

```
class Menu : Core::Group
{
  property string Caption = "Welcome to $ProductName";
  property string Version = "Ver:" + string( $Version );

  method object HandleEvent( arg aEvent )
  {
    var Core::KeyEvent keyEvent = (Core::KeyEvent)aEvent;

    if ( keyEvent != null )
      switch ( keyEvent.KeyCode )
      {
        case $MenuKey  : OpenMenu( $MenuSize );
        case $CloseKey : CloseMenu();
        default :;
      }

    return null;
  }
}
```

Except for the above-mentioned limitation, each macro is evaluated and replaced by the preprocessor without any exception — even if a macro has been used within a `string`, `char`, `rect`, `point`, `color`, etc. literal:

```
var string text    = "$ProductName";
var rect   bounds  = <0,0,$Size>;
var color  bgColor = #$Red$Green$BlueFF;
...
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Because the macro placement occurs even within `string` and `char` literals, and because the '`$`' sign is used exclusively by the preprocessor, the '`$`' character itself must be indicated by a sequence of two '`$$`' signs. The preprocessor replaces each sequence of 2 '`$$`' signs with a single '`$`' character, without evaluating any macros in the process:

```
var string price1 = "100 $";  // Error!
var string price2 = "100 $$"; // Ok == "100 $"
```

A further limitation concerns the nesting of macros. Unlike with the C, C++ preprocessor, Chora macros can't be nested. The nesting of macros would mean that in the definition of a macro, a different macro would be used. That is not allowed, and causes a Chora compiler error:

```
profile Target1 : Tara.Win32.RGBA8888
{
  macro Size = 640,480;
  macro Rect = <0,0,$Size>;
}
```

Aside from allowing user-defined macros, the Chora preprocessor makes its own set of predefined macros available. These macros can be used as needed, without being defined explicitly in a profile:

| Makro name | Description |
|---|---|
| `$time` | `$time` is always replaced by a character string consisting of the date and time of the currently running code generation. For example:<br>`'04.02.2003 12:17:59'` |
| `$line` | `$line` is always replaced by the line number of the code in which the `$line` macro is found.<br>The numbering of the lines of code is limited only to methods and the inline code definitions, and begins each time with the line number 1. For example:<br>`'59'.` |
| `$path` | `$path` is always replaced by the full name of the currently evaluated method or inline code definition. For example:<br>`'Core::Group.HandleEvent'`<br>`'Core::TimerInline'` |
| `$version` | `$version` is always replaced by the version number of the used Chora compiler. For example:<br>`'3.30'.` |
| `$profile` | `$profile` is always replaced by the name of the profile currently selected for the code generation. For example: |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| | `'Win32'` |
| `$platform` | `$platform` is always replaced by the name of the platform package used in the currently selected profile. For example:<br>`'Tara.Win32.RGBA8888'` |
| `$prototyper` | The `$prototyper` macro is replaced by either the value `true` or the value `false`, depending on whether the code is generated for the target system or for the Embedded Wizard prototyping environment:<br>`'true'  // prototyping environment`<br>`'false'  // target system` |
| `$composer` | The `$composer` macro is replaced by either the value `true` or the value `false`, depending on whether the code is generated for the Embedded Wizard Composer editor, or not:<br>`'true'  // composer (editor) mode`<br>`'false'  // prototyper or target system` |
| `$verbose` | The `$verbose` macro is replaced by either the value `true` or the value `false`, depending on the `Verbose` attribute of the currently selected profile. This attribute controls the amount of comments included into the generated code. If this attribute is `true`, the Code Generator generates more detailed comments. If this attribute is set `false`, only the most important comments are included. The effect of this attribute depends on the used Code Generator:<br>`'true'  // verbose mode is on`<br>`'false'  // verbose mode is off` |
| `$optimization` | The `$optimization` macro is replaced by the value `None`, `Low`, `Medium` or `High`, depending on the value of the `Optimization` attribute of the currently selected profile. This attribute controls the level of optimization during the code generation. Depending on the value of this attribute, the Code Generator may perform different code simplification and elimination steps. Following optimization levels are available:<br>`'None'  // Optimization is disabled`<br>`'Low'  // Simple optimization`<br>`'Medium'  // Medium level of optimization`<br>`'High'  // High level of optimization` |
| `$obfuscation` | The `$obfuscation` macro is replaced by either the value `true` or the value `false`, depending on the `Obfuscation` attribute of the currently selected profile. |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| | The obfuscation attribute controls the format of the generated code. If this attribute is `true`, the Code Generator will try to compress and obfuscate the generated code. If this attribute is set `false`, the code is not modified.<br><br>The effect of this attribute depends on the used Code Generator. If the generator doesn't support the code obfuscation, then the attribute has no effect on the generated code:<br><br>`'true'  // obfuscation mode is on`<br>`'false'  // obfuscation mode is off` |
| `$ApplicationClass` | The `$ApplicationClass` macro is replaced by the value of the `ApplicationClass` attribute of the currently selected profile. This attribute defines the root class of the entire GUI application. |
| `$ApplicationTitle` | The `$ApplicationTitle` macro is replaced by the value of the `ApplicationTitle` attribute of the currently selected profile. This attribute defines the optional GUI application title as a string literal. |
| `$ApplicationName` | The `$ApplicationName` macro is replaced by the value of the `ApplicationName` attribute of the currently selected profile.<br><br>This attribute specifies an optional application name as an identifier (e.g. `DashboardApp`). This attribute allows you to distinguish the application within an environment executing simultaneously multiple Embedded Wizard applications.<br><br>The effect of this attribute depends on the platform package used for the code generation. If the generator doesn't support the code obfuscation, then the attribute has no effect on the generated code. |
| `$ScreenSize` | The `$ScreenSize` macro is replaced by the value of the `ScreenSize` attribute of the currently selected profile. This attribute defines the desired size of the screen in pixels as a Chora `point` literal, e.g. `<720,576>`. |
| `$Clut` | The `$Clut` macro is replaced by the value of the `Clut` attribute of the currently selected profile. This attribute defines the name of the Color Lookup Table (CLUT) file containing the definition of colors used within this GUI application. The CLUT is used for `Index8` platforms only. |

**Tabelle 8-1**

The final limitation concerns the use of reserved preprocessor directives as macros. The preprocessor reserves certain keywords, which it evaluates as special directives. These directives, like the macros, begin with the character '`$`', but must not be confused with the macros. Thus one may not use the following macros: `$if`, `$else`, `$elseif`, `$endif`, `$error`, `$warning`.

You can read more about the preprocessor directives in "Preprocessor directives" (chapter 8.2).

## 8.2 Preprocessor directives

The preprocessor directives allow the Chora compiler to exclude explicitly chunks of code from the code generation. For example, by evaluating a condition in the `$if` directive, the preprocessor decides whether the following chunk of code will pass the preprocessor or it will be ignored:

```
method void Mute()
{
  $if $platform == Tara.Win32.RGBA8888
    native
    {
      __send_i2c( Audio, OFF );
    }
  $endif
}
```

Only if the condition `$platform == Tara.Win32.RGBA8888` is fulfilled the immediately following code lines, up to the `$endif` directive, will pass the preprocessor. If the condition is not fulfilled, the Chora compiler ignores all these lines of code until the `$endif` directive has been evaluated.

The `$if` and `$endif` directives may only be used by pairs. That is to say, each `$if` directive must be closed by a `$endif` directive. Nesting of directives is not allowed, and leads to a Chora error message:

```
$if $platform == Tara.Win32.RGBA8888
  $if $composer == false      // Error!
    ...
  $endif
  $if $composer == true       // Error!
    ...
  $endif
$endif
```

The preprocessor expects that the expression in the condition of a `$if` directive will result in a value of either `true` or `false`. Using logical operators and parentheses, complex conditions can be created. The Chora preprocessor supports the following operators:

| Operator name | Operand data type | Description |
|---|---|---|
| ! | ! bool | Logical negation. The result is the logical negation of the value of the operand. `false` if the operand is `true` and `true` if the operand is `false`.<br>`$if !( $profile == Win32 )`<br>`...` |

| | | |
|---|---|---|
| | | `$endif` |
| `&&` | `bool && bool` | Logical 'and'. The result is true if both operands are true.<br>`$if $composer && $prototyper`<br>`  ...`<br>`$endif` |
| `\|\|` | `bool \|\| bool` | Logical 'or'. The result is true if one of the operands is true.<br>`$if ( $Ver == 0 )\|\|( $Ver == 1 )`<br>`  ...`<br>`$endif` |
| `==` | `Op1 == Op2` | Equal to. This operator returns `true`, if both operands `Op1` and `Op2` result in exactly the same character string.<br>`$if abc == abc`<br>`  // Ok, pass the preprocessor`<br>`$endif`<br><br>`$if abc == ABC`<br>`  // will be ignored`<br>`$endif` |
| `!=` | `Op1 != Op2` | Not equal to. This operator returns `true`, if the two operands `Op1` and `Op2` do not result in exactly the same character string.<br>`$if abc != abc`<br>`  // will be ignored`<br>`$endif`<br><br>`$if abc != ABC`<br>`  // Ok, pass the preprocessor`<br>`$endif` |

**Tabelle 8-2**

By using these operators, a complex condition can be formulated, which is fulfilled only when the partial conditions, for example `$profile == Win32` and `$composer == true`, are also fulfilled:

```
$if ( $profile == Win32 ) && ( $composer == true )
  // Pass this code if both conditions are true only.
  ...
$endif
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

Before the preprocessor can evaluate the expression in a `$if` condition, all macros contained in the expression are replaced by their contents. → See "Macro evaluation" (chapter 8.1). This makes it very easy to control the code generation conditionally through the definition of macros. For example, the following chunk of code will pass the preprocessor only if the `Debug` macro in the currently selected profile contains the character string `true`:

```
$if ( $Debug == true )
  trace Menu.Visible, Menu.Bounds, Menu.Caption;
$endif
```

In this case, the expression can be further simplified if the `Debug` macro contains either the character string `true` or `false`:

```
$if $Debug
  trace Menu.Visible, Menu.Bounds, Menu.Caption;
$endif
```

In comparisons using the `==` or `!=` operator, one of the both operands may contain special wildcards '`*`' and '`?`'. The character '`*`' stands for an arbitrary number of any characters, including an empty string. The '`?`' wildcard, on the other hand, stands for exactly one character. These wildcards are used like patterns in comparing character strings:

```
$if $platform == Tara.Win32.*
  // Pass this code for Win32 platforms only
$endif
```

Only if the content of the `$platform` macro begins with the string `Tara.Win32.` is the `$if` condition shown above fulfilled, and the following chunk of code can pass. This condition applies in equal measure to the strings `Tara.Win32.Index8` and `Tara.Win32.RGBA8888`, etc.

The wildcards are very practical for flexibly comparing character strings. For example, when the major version number of the Chora compiler must be evaluated:

```
$if $version == 1.?*
  // Chora Compiler Version 1.00, 1.13, 1.2, etc.
$endif
```

Two additional directives, `$else` and `$elseif`, simplify the conditional code generation. Any chunks of code after the `$else` directive are considered by the Chora compiler only if the corresponding `$if` condition is not fulfilled:

```
$if $platform == Tara.Win32.RGBA8888
  // Pass this code if the condition is true only.
  ...
$else
  // Pass this code if the condition is false only.
  ...
$endif
```

The `$elseif` directive combines `$else` and `$if`, so that multiple chunks of code may successively pass the preprocessor, each under a different condition:

```
$if $platform == Tara.Win32.RGBA8888
   // Pass this code if the condition is true only.
   ...
$elseif $platform == Tara.Win32.Index8
   // Pass this code if the second condition is true.
   ...
$elseif $platform == Tara.Win32.RGBA4444
   // Pass this code if the third condition is true.
   ...
$else
   // Pass this code if all conditions are false
   ...
$endif
```

Directives may only be used at the beginning of a line of code. Only one directive is allowed per line of code. After the directive, no further Chora statements may follow in the same line of code. The following example shows what is not allowed:

```
$if ( $Debug ) trace Menu.Visible;  $endif
```

This example causes a Chora compiler error.

The Chora preprocessor provides also two additional directives: `$error` and `$warning`. If evaluated, these directives force the Chora compiler to report an error or a warning. These directives expect a message text, which is then displayed in the compiler output:

```
$if $FAST && $SMALL
  $error Invalid configuration settings
$endif
```

## 8.3 Control directives

The Chora programming language supports the following control directives:

| Directive name | Description |
|---|---|
| `$output` | Die `$output` directive determines whether the definition immediately following it is considered during the code generation or simply ignored.<br>The simplest version of the `$output` directive expects a single `bool` argument, either `true` or `false`. In the following example the both `$output` directives force the code generation for the class and the variable:<br>`// Generate code for this Menu class`<br>`$output true`<br>`class Menu : Core::Group`<br>`{`<br>`  $output true`<br>`  var string Caption = "Sound";`<br>`}` |

| | |
|---|---|
| | The `$output` directive accepts also profile names. In this case the code generation is forced if the affected profile was selected for the code generation only. For example the following method should be generated for the `Win32` target system only - the name of the profile `Win32` appears in the `$output` directive:<br><br>**`$output Win32`**<br>`void LoadAudioSettings( arg bool aMode )`<br>`...`<br><br>To build complex conditions, the `$output` directive accepts a list of profile names. These operands have to be separated by comma signs. In this case the `$output` directive is fulfilled if one of the operands is fulfilled. For example the following method should be generated for the `Win32` or `Linux` target system only:<br><br>**`$output Win32, Linux`**<br>`void LoadAudioSettings( arg bool aMode )`<br>`...`<br><br>If the condition in the `$output` directive is not fulfilled, the following definition is not generated, unless there is a dependency from other parts of the project to this definition. In this case the affected definition is automatically involved into the code generation. This behavior can't be suppressed.<br><br>If not explicitly specified, the default value for this directive is set `false` for all members.<br><br>For additional details see "Optimization" (chapter 13). |
| `$variant` | The `$variant` directive specifies the variant condition for a class, constant or resource variant. This directive affects only the definition following immediately.<br><br>The simplest version of the `$variant` directive expects a single `bool` argument, either `true` or `false`. If the argument is `true`, the affected variant substitutes its origin class, constant or resource permanently – it is so called 'static' variant. In the following example the used `$variant` directive forces the class `Menu::MenuItem` to be substituted by its static variant `AquaItem`:<br><br>`// The class variant AquaItem substitutes`<br>`// its origin class Menu::MenuItem`<br>**`$variant true`**<br>`vclass AquaItem : Menu::MenuItem`<br>`{`<br>`  ...`<br>`}`<br><br>The `$variant` directive accepts also profile and style names. In this case the variant is used only, if the specified profile or |

| | |
|---|---|
| | style is currently selected. The profiles can be selected during the code generation only. This provides a more flexible kind of static variants, which are resolved already during the code generation. For example the following constant variant is used in the `Win32` target system only:<br><br>`$variant Win32`<br>`vconst Caption : Dialogs::Caption`<br>`{`<br>`  Default = "Modified Caption";`<br>`}`<br><br>The usage of style names in the `$variant` directive provides dynamic variants. The affected variant is used, if the specified style is selected at the runtime only.<br><br>The `$variant` directive accepts also a list of comma-separated profile and style names. In this case the variant condition is fulfilled if one of the specified profiles or styles is selected:<br><br>`$variant Aqua, Graphite`<br>`vclass NiceItem : Menu::MenuItem`<br>`{`<br>`  ...`<br>`}`<br><br>If not explicitly specified, the default value for this directive is set `true`. For more details see "Variants" (chapter 4.8) and "Usage of Variants" (chapter 10). |
| `$multilingual` | The `$multilingual` directive controls the code generation for the automatic re-initialization of class objects as a result of the language switch. This directive affects the members of the following class definition only.<br><br>`$multilingual` expects an argument, either `true` or `false`:<br><br>`// Enable automatic re-initialization.`<br>`$multilingual true`<br>`class Menu : Core::Group`<br>`{`<br>`  ...`<br>`}`<br><br>If not explicitly specified, the default value for this directive is set `false`. For more details see "Language selection" (chapter 9). |
| `$rect` | The `$rect` directive defines the position and the size of a brick symbol of the definition that immediately follows. These dimensions are used internally in the Embedded Wizard to arrange the brick symbols in the composer editor.<br><br>`$rect` expects a rectangle as a `rect` literal:<br><br>`$rect <10,10,160,50>`<br>`property string Caption;` |


Copyright TARA Systems GmbH
www.embedded-wizard.de

| | |
|---|---|
| `$reorder` | The `$reorder` directive is used in the definition of classes to change the Z order of inherited and new class members. The Z order plays an important part in the initialization of objects. → See "'new' operator" (chapter 7.6).<br><br>`$reorder` allows to change the Z order of a member. For this purpose `$reorder` expects the name of the affected member and the desired displacement number:<br><br>**`$reorder Caption 2`**<br><br>The `$reorder` can be placed anywhere inside the definition of a class.<br><br>Generally it may be very confused to determine the correct Z order of a member defined and derived in several classes. The Z order of a member is affected by other members defined in the base classes, by the `$reorder` directives. For this purpose `$reorder` directives are prepared automatically by Embedded Wizard after a class has been modified in the Composer window. |
| `$version` | The `$version` directive must, at the beginning of any project or unit, determine the version of the Chora syntax used:<br><br>**`$version`** `5.0`<br><br>`...`<br><br>If this directive is missing, the Chora compiler assumes that the version of the used Chora syntax is 1.0. It is important to determine the correct version of the used syntax because of the modifications in syntax between the version 1.0 and 2.0. If this directive is used correctly, the Chora compiler is able to evaluate files containing Chora 1.0 and 2.0. Mixing of files with different syntax versions is possible. |
| `$include` | The `$include` directive specifies an include file of a split unit. This directive is valid within the `.DIR` file only.<br><br>The `$include` directive is followed by the name of the affected include file, without the extension:<br><br>**`$include Timer`**<br><br>More details can be found in "Unit file (EWU)" (chapter 4). |

**Tabelle 8-3**


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 9 Language selection

The Chora programming language supports the development of multilingual GUI applications. A multilingual GUI application contains GUI definitions with texts, images, etc., for various languages. At the application's runtime, it is then possible to switch between the different languages. A German user, for example, will select the German language, while a French user will prefer French.

The development of a multilingual GUI application requires that all desired languages are listed explicitly in the project file. This is done using the keyword `language`:

```
language German;
language French;
language Greek;
language Default;
```

Note that the language `Default` has a special importance and must always be defined. This `Default` language stands for all parts of the GUI application that are not language-dependent. A precise description of how languages are defined can be found in the chapter "Languages" (chapter 3.3).

Chora's multilingual function is limited to the definition of constants and resources. This means that only constants and resources may have multiple language-dependent contents. For example, a constant `string` with text in German and English:

```
const string Caption =
(
  German = "Hallo Welt!";
  Default = "Hello World!";
);
```

Multilingual resources are defined in a very similar way. A single attribute of a resource can then adopt a number of different values, for example in order to define a bitmap resource `Logo` with two different images:

```
resource Resources::Bitmap Logo
{
  attr bitmapfile FileName =
  (
    German  = GermanBanner.png;
    Default = EnglishBanner.png;
  );
}
```

Note the use of the `Default` language in the above definition. By means of `Default` the language-independent content of a constant or resource is set. Any languages not explicitly listed in the definition will consequently use the content of `Default`. Any multilingual constant or resource must define at least the `Default` language variant. If `Default` is missing, a Chora error is caused. Further details on the definition of constants and resources are found in the chapters "Constants" (chapter 4.2) and "Resources" (chapter 4.3).


Copyright TARA Systems GmbH
www.embedded-wizard.de

When a multilingual constant or resource is accessed at runtime, the currently selected language is evaluated. This selected language determines which language variant of the desired constant or resource is returned. For example, when the `German` language is selected, the constant `Caption` defined above returns the string `"Hallo Welt"`. For all other languages (`Default`, `French`, `Greek`), the string `"Hello World"` is returned.

The currently selected language is controlled by the global built-in variable `language`. `language` is one of the few global variables in the Chora programming language. To switch the language, the name of the desired language must be assigned to this global `language` variable:

```
if ( SwitchToGermanCondition )
  language = German;

if ( SwitchToGreekCondition )
  language = Greek;

...
```

The content of the `language` variable can, of course, also be evaluated in an expression, for example in order to query the currently selected language:

```
// Toggle between Default and German language
if ( language == German )
  language = Default;

if ( language == Default )
  language = German;

...
```

During the development, the language can be selected in a convenient way directly in the Language combo box of the Embedded Wizard IDE:

Aside from the global built-in variable `language` there is also an instant data type of the same name, `language`. This data type is used whenever variables, properties or arrays must be defined, in order to store there the name of a language:

```
// Store the currently selected language in the temporary
// variable.
var language tmp = language;

// switch to the Default language
language = Default;

... Do something here ...;

// Restore the previously selected language
language = tmp;
```

An assignment to the variable `language` changes only the currently selected language. This has no effect on constants or resources that have already been evaluated. Changing languages has an effect only when constants or resources are re-evaluated again:

```
var string text1;
var string text2;

language = Default;
text1 = Sample::Caption;

language = German;
text2 = Sample::Caption;

trace text1;         // text1 = "Hello World"
trace text2;         // text2 = "Hallo Welt"
```

The Chora compiler is able to generate the necessary re-initialization code. The language switch will then trigger the objects to re-evaluate theirs language dependent initialization expressions and to redraw themselves. This is done automatically in a way fully transparent to the programmer.

To use the automatic re-initialization, the class of affected objects should be signed as 'multilingual'. This is done by the `$multilingual` directive:

```
$multilingual true
class Menu : public Core::Group
{
  object Views::Text Caption
  {
    preset String = TheUnit::TheCaption;
    preset Color  = #FF0000FF;
  }
}
```

In this example all objects of the `Menu` class are able to reinitialize themselves. Assumed the preset `String` was initialized with the value of a multilingual constant `TheUnit::TheCaption`, the affected initialization expression will be re-evaluated automatically after the languages has changed. The second initialization expression for the preset `Color` is not re-evaluated, because there are no language dependent constants or resource operands.

The Chora compiler analyses each initialization expression end generates the additional re-initialization code only for expressions, which contain at least one multilingual constant or resource operand.

The `$multilingual` directive affects the following class definition only. If this directive is missing or it is set `false`, no additional code for the automatic re-initialization of this class members is generated. The directive does not affect base or derived classes.

If necessary an initialization expressions may be excluded from the automatic re-initialization. To do this, the expression have to start with the '`%-`' (percent minus) init operator. In this manner the following expression is ignored by the Chora compiler and will not be re-initialized, even if the expression is multilingual:


Copyright TARA Systems GmbH

```
$multilingual true
class Menu : public Core::Group
{
  object Views::Text Caption
  {
    preset String = %-TheUnit::TheCaption;
  }
}
```

In the same way the Chora compiler can be forced to generate re-initialization code for a non multilingual expression. In this case the '`%+`' (percent plus) init operator have to be used:

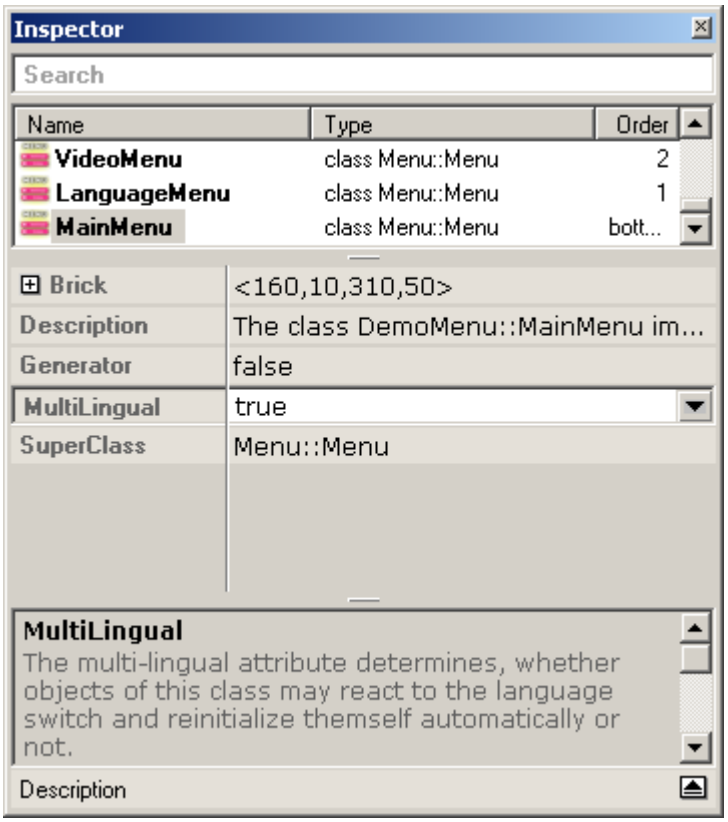
```
$multilingual true
class Menu : public Core::Group
{
  object Views::Text Caption
  {
    preset Color  = %+#FF0000FF;
  }
}
```

The order of the generated re-initialization expressions corresponds to the Z order of the affected members. This is the same order, which is used to generate the first initialization code. For more details see "'new' operator" (chapter 7.6).

Similar to the first initialization, at the end of the re-initialization the `ReInit()` re-constructor is invoked. You can implement the `ReInit()` method in order to do special re-initializations, which are not done by the automatic generated re-initialization code. Unlike the automatic re-initialization, the `ReInit()` method is always invoked, even if the affected class is signed as non multilingual. For more details see "ReInit re-constructors" (chapter 4.1.5.7).

In the Embedded Wizard IDE the value for the multilingual directive is defined directly in the Inspector Window:


Copyright TARA Systems GmbH
www.embedded-wizard.de




Beside the language-dependent constants and resources, Chora also provides variants → see "Variants" (chapter 4.8) and "Usage of Variants" (chapter 10).


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 10 Usage of Variants

Variants are a unique technique that was introduced into the Chora programming language in order to simplify the customization approach of existing Chora programs. Using this feature, the appearance and the behavior of an GUI application can be modified and extended without the necessity of any modifications on the existing software.

Within an GUI application multiple variants can be implemented and selected during the code generation or dynamically at the runtime. This feature provides the basis for GUIs with different appearances. Skins and themes can be realized and the GUI can be adapted to target systems with different features and screen resolutions.

The variants are based on the concept of object-oriented programming. By deriving a variant from a class, the implementation of the class can be adapted and extended as desired by the developer. The methods of the class can be overridden, the initialization values of inherited members can be changed and new members can be added to the class. All these adaptations are possible without any modifications of the origin class → see "Class Variants" (chapter 4.8.1).

In the same way, constants and resources are adapted. Every constant and resource can be overridden by variants. The colors, bitmaps, fonts, menu strings, etc. used within the GUI application can then be substituted by new values. In case of language-dependent constants and resources, new languages can be added. All this is done without any modifications on the origin constant or resource → see "Constant Variants" (chapter 4.8.2) and "Resource Variants" (chapter 4.8.3).

The both concepts of multivariant and multilingual applications do complement one another. Consequently a very powerful and unique technique is available, where the appearance, the behavior and the language-dependent strings, images, colors, etc. can be defined and switched at the runtime. The handling of multilingual applications is described more detailed in "Language selection" (chapter 9).

A variant does never exist alone – it always belongs to the class, constant or resource, it has been derived from. When the origin class, constant or resource is used in the GUI application, the Embedded Wizard will replace it with the appropriate variant automatically. The following example demonstrates the definition of a variant `MyCaption` of the origin constant `Menu::Caption`:

```
vconst MyCaption : Menu::Caption = "Hello variant!";
```

Although the name of the variant `MyCaption` is well known, it is not allowed to use it directly in an expression. The constant can be accessed through its origin name `Menu::Caption` only. Anyway, the origin constant is influenced by the variant – the default value of the constant is overridden, so at the runtime the evaluation of the origin constant will result in the string `"Hello variant!"`:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
var string text;

// This expression will cause Chora compiler error,
// because the variant can't be accessed directly.
text = MyMenu::MyCaption;

// Access the constant. Since the constant is
// overridden by a variant, the expression results in
// the value of the variant itself:
text = Menu::Caption;   // text = "Hello variant!"
```

The variant defined in the example above is called static, because the variant substitutes the origin constant permanently - it can be switched neither during the code generation nor at the runtime – it is fixed.

More flexible variants expect an additional variant condition directive. This directive determines when the affected variant will be used. For example, if the variant `MyCaption` should be used in case of selected `Win32` profile only, the name of the profile should be specified in the variant condition:

```
$variant Win32
vconst MyCaption : Menu::Caption = "Hello variant!";
```

In this case the origin constant is not substituted unless the code is generated for the profile `Win32`. If the profile specified in the directive does not exist in the project, it is ignored and the affected variant is not generated. A variant can also depend on several profiles. If one of the profiles specified in the directive is selected for the code generation, the variant does substitute the origin constant:

```
$variant Win32, Win32_Debug
vconst MyCaption : Menu::Caption = "Hello variant!";
```

As long as a variant depends on profiles, it still remains static - the variant will be selected already during the code generation and it can't be switched at the runtime anymore. Only dynamic variants can be switched at the runtime.

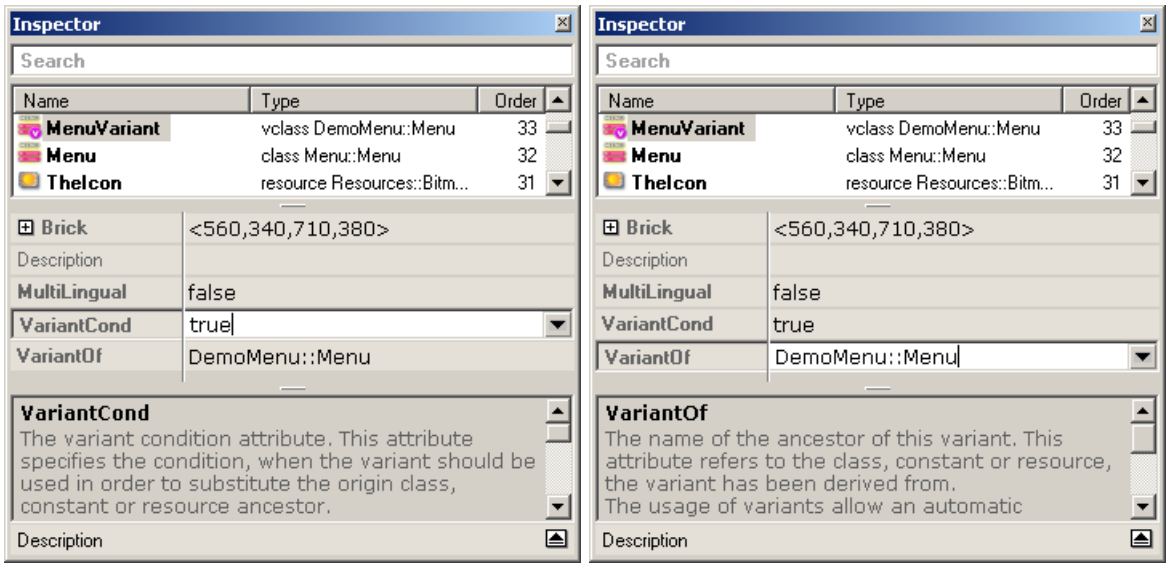
Dynamic variants do always depend on styles. In order to select the variant, the style specified in its variant condition must become active. For example the following constant variant `MyCaption` will substitute its origin constant `Menu::Caption`, if the style `Aqua` will become active at the runtime:

```
$variant Aqua
vconst MyCaption : Menu::Caption = "Hello variant!";
```

Please note, there is no differences in the syntax of the `$variant` directive for static and dynamic variants. In both cases, the directive expects identifiers separated by comma signs. The specified identifiers are evaluated during the code generation. At this time point, the Chora compiler distinguishes between profiles and styles - this results in static or dynamic variants.

In the Embedded Wizard IDE the variant condition and the ancestor of the variant are defined directly in the Inspector Window:


Copyright TARA Systems GmbH




When working with dynamic variants, all desired styles have to be listed previously in the project file. This is done using the keyword `style`:

```
style Aqua;
style WideScreen;
```

For more details see "Styles" (chapter 3.4).

The styles can be activated and deactivated at the runtime by assigning the desired styles set to the Chora global built-in variable `styles`. In the Chora syntax, a styles set literal is always enclosed in brackets. The following example demonstrates how the both styles `Aqua` and `WideScreen` are activated simultaneously:

```
styles = [Aqua, WideScreen];
```

To deactivate all styles, simply assign an empty set to the `styles` variable:

```
styles = [];
```

The assignment to the variable `styles` does affect the following operations only. Expressions evaluated already in the past are no more affected by the switch. The following example demonstrates it. The both variables `text1` and `text2` are initialized with the value of the constant `Menu::Caption`. `text1` variable will receive and keep the origin, non substituted value of the constant. The second expression is evaluated after the style `AquaLook` has become active. In this case the second variable `text2` will receive the value overridden in the variant `MyCaption`:

```
var string text1;
var string text2;

// Ensure, the styles are deactivated and get the value
// of the constant.
styles = [];
text1 = Menu::Caption;

// Activate the style and get the value of the constant.
styles = [Aqua];
text2 = Menu::Caption;
```

The content of the `styles` variable can, of course, also be evaluated in an expression, in order to query the currently active styles. For this purpose Chora implements several operators. For example the instant method `contains()` determines, whether a styles set contains another styles set. If this is fulfilled, the method returns `true`:

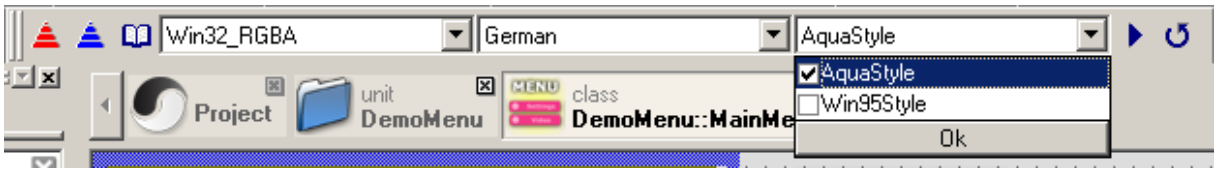
```
// Toggle the Aqua style in the global styles
// variable.
if ( styles.contains( [Aqua]))
  styles = styles - [Aqua];
else
  styles = styles + [Aqua];
```

In the above example the operators '+' and '–' have been used to build a union or a difference of two styles sets. A more simple way to toggle a style can be implemented by using the '^' exclusive OR operator:

```
styles = [Aqua] ^ styles;
```

The supported operators are described more detailed in "Binary instant operators" (chapter 7.2).

During the development, the styles can be activated and deactivated in a convenient way directly in the Styles combo box of the Embedded Wizard IDE:




Aside from the global variable `styles` there is also an instant data type of the same name, `styles`. This data type is used whenever variables, properties or arrays must be defined, in order to store there the names of active styles:

```
// Store the currently active styles in the temporary
// variable.
var styles tmp = styles;

// switch the styles
styles = [WideScreen];

... Do something here ...;

// Restore the previously active styles
styles = tmp;
```

The handling of multivariant classes and resources is performed in the same way as this was demonstrated in the examples above. Assumed, the class `Menu::MenuItem` is multivariant and the variant depends on the style `Aqua`, then two different objects of the same class can be created at the runtime:

```
var Menu::MenuItem object1;
var Menu::MenuItem object2;

// Ensure, the styles are deactivated and create the
// object.
styles = [];
object1 = new Menu::MenuItem;

// Activate the style and create an object of the variant
styles = [Aqua];
object2 = new Menu::MenuItem;
```

After the creation, the both objects retain their derivation hierarchy until the objects are disposed. Anyway, because the objects are instantiated from different class variants, they may and will behave differently! In these class variants new members could be added and the logic of inherited methods could be overridden. Anyway, these objects can coexist simultaneously and the Embedded Wizard can handle them properly. This is a special feature of Chora.

The one restriction in the usage of variants is, that the variants itself can never be accessed directly within an expression. This is because the variants are not handled as independent definitions – they always belong to the origin definition. If possible the variants are combined together with their ancestors or they are completely eliminated. In the case of a class variant, it is not possible to access its new added members, from the outside of the variant. These members are always protected (hidden).

The second restriction concerns the definition of multiple variants of a single class, constant or resource. If two variants do depend on the same variant condition, the Embedded Wizard is not able to distinguish between them – a Chora compiler error is reported.

In case of dynamic variants, the selection of the appropriate variant depends on the styles activated at the runtime. This could cause unexpected behavior at the runtime, when two or more variants of a single origin definition do depend on these active styles. For this purpose the selection of dynamic variants is always performed regarding to the Z-order of the styles – only the variant is selected, whose Z-order is lower.

The concept of variants is very powerful. Beside the derivation of variants from origin classes, constants, resources and autoobjects, it is also possible to derive variants from other variants. In this manner, complex variant hierarchy can be build. Furthermore, in the class hierarchy each class can be derived by own variants. While the ordinary class hierarchy is only one dimensional, the variants extend it by a kind of a second dimension. Figure 10–1 demonstrates it:


Copyright TARA Systems GmbH
www.embedded-wizard.de

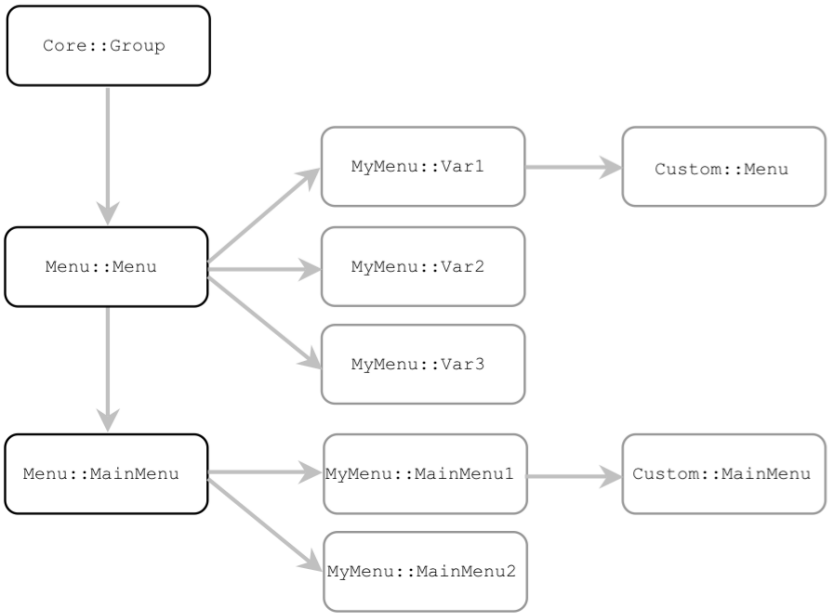

Core::Group
Menu::Menu
Menu::MainMenu
MyMenu::Var1
MyMenu::Var2
MyMenu::Var3
MyMenu::MainMenu1
MyMenu::MainMenu2
Custom::Menu
Custom::MainMenu

**Figure 10–1**

# 11 Garbage collection

Garbage collection is a special feature of the Chora programming language. It relieves the programmer of the responsibility of having to explicitly delete unused objects. The Garbage Collector determines unused objects and automatically takes care of their disposal. This eliminates one of the most-feared sources of programmer error, for thanks to garbage collection it is no longer possible for dynamically created objects to be destroyed more than once or to remain undesirably in memory as memory leaks.

How the Garbage Collector works actually depends on the target system. In general, however, a Garbage Collector works in 2 phases:

- Phase 1: Mark. During the first phase, the Garbage Collector attempts to locate and mark all objects that are still needed. All objects thus marked are then protected from deletion. During the marking phase, the Garbage Collector traces all the connections between the objects and thus marks every object it can reach. The process ends when all reachable objects have been marked. The starting point for the route taken by the Garbage Collector is a so-called root set, that is, one or more objects that exist during the entire lifetime of an GUI application and which correspondingly may not be deleted.
- Phase 2: Sweep. In the second phase, all objects not marked are destroyed. Here, the Garbage Collector wanders through the entire memory heap and searches for objects that were not marked during the first phase. These objects are then destroyed.

Objects are also destroyed in 2 steps:

- First, the object in question must be de-initialized. To do this, the Garbage Collector invokes the object's destructor. The destructor gives the object a chance to release all system resources previously reserved or to restore the state of the hardware the object controls. Details on destructors can be found in the chapter "Done destructors" (chapter 4.1.5.6).
- When the destructor is finished, the memory taken up by the object is released and can be used for other objects.

An important feature of the Garbage Collector is the 100% certainty that only unused and unreachable objects are disposed of. An object is unreachable when there is no any references in the entire GUI application that could refer to that object. Only then can the object never again be accessed, and be disposed of as garbage.

For this reason, it is important to understand that under no circumstances may one use one object's destructor to access other objects. In some cases, that makes the object that is being destroyed reachable again. At such time, the object's ongoing destruction can no longer be avoided. The result is an unpredictable malfunction in the GUI application. The following code demonstrates what not to do in a destructor:


Copyright TARA Systems GmbH
www.embedded-wizard.de

```
method void Done()
{
  // By the following assignment this object becomes
  // reachable again!!!! Don't do this!!!
  AnOtherReachableObject.Link = this;
}
```

A further issue relating to garbage collection is the point in time when it is started. In most target systems, this point is random and thus not predictable. This means that an object's destruction can be delayed. Although the Garbage Collector guarantees that all unused objects are disposed of, the time of the disposal is not known.

This limitation is not actually a problem. In a few cases, however, it can become problem when an object is supposed to release a particularly scarce system resource in the destructor. The delay can result in a temporary unavailability of important system resources. In such a very rare case, we recommend implementation of a `Close()` or `Free()` method which, explicitly invoked, can release the scarce system resource and do it long before the Garbage Collector could dispose the object.

Because destructors are used almost exclusively in driver classes, the typical GUI application programmer is spared the consequences described here.


Copyright TARA Systems GmbH
www.embedded-wizard.de

# 12 Comments

Unlike other programming languages, in Chora comments are fully valid syntactic elements. They are used as the inline documentation for languages, units, profiles, constants, resources, classes, methods, properties, etc.. This documentation is displayed in Embedded Wizard application in the 'Short Info' area of the 'Inspector' window. Each time a member is selected in the 'Composer' window, the associated documentation will appear in the 'Short Info' area immediately.

The more important use case for the comments is the automatic generation of the documentation file. Embedded Wizard evaluates the content of your project, the dependencies between the classes, constants, resources, etc. and stores it together with the associated inline documentation within a set of HTML files. In the second step a single MS-Windows help file is created. This help file contains detailed documentation of your project and it can be used or distributed as the 'software reference manual'. For more details, how to generate the documentation file see the 'Embedded Wizard User Manual'.

During the generation of the documentation file the content of all affected comments is preprocessed by the Embedded Wizard. In this manner cross references are added and additional formatting of the text of the inline documentation occurs. For this preprocessing following rules are used:

1. While formatting, all text lines of the inline documentation are converted in HTML paragraphs. Within the comment a single text line is always terminated by the carriage return. To suppress the carriage return a `'\'` backslash sign should be used at the end of the text line. In this case multiple lines are concatenated together to a single text block, which will appear within a single HTML paragraph. The HTML browser is then responsible for the text flow and the line wrapping occurs automatically.

   ```
   // This should be a very long, long text line, so it has\
   // to be continued in the next line to appear within a\
   // single paragraph.
   // This line appears in the next paragraph.
   class Menu : Forms::Form
   {
     ...
   }
   ```

2. Identifiers started with the `'$'` dollar sign are recognized as potential macros. In the case the identifier is a valid macro name, defined in your project, a cross reference to this macro description is added to the text. For example the following inline documentation will be extended by a cross reference to the description of the macro `$Size`:

   ```
   // Set the bounds of the menu to the size of the screen\
   // defined in the macro $Size.
   inherited property Bounds = rect(<0,0>, $Size );
   ```


Copyright TARA Systems GmbH

3. Identifiers separated by the `'::'` double colon are recognized as a potential name of a class, constant, resource or enumeration. In the case the name belongs to an existing definition within your project, a cross reference to the description of this definition is added to the text. For example the following inline documentation will be extended by a cross reference to the description of the class `Core::View`.

```
// Override the Draw() method of the super class\
// Core::View in order to be able to fill the background\
// with the blue color.
inherited method Draw()
{
  ...
}
```

4. Identifiers started with the `'@'` at sign are recognized as potential class, unit or project members. Depending on the context the `'@'` at sign is used in, the class hierarchy, the unit or the list of languages, profiles and styles is seek for a member, which name matches the given identifier. For example the following inline documentation will be extended by a cross reference to the description of the method `HandleEvent()`.

```
// The method DoSomething() is called from the method\
// @HandleEvent()each time a key down event arrives.
method void DoSomething()
{
  ...
}
```

   While seeking for the member with the given identifier the inheritance hierarchy of the affected class is evaluated till one of the base classes contains the description of the desired member.

   If the desired member is not defined in any of the base classes, the members of the superior unit are evaluated. If this operation is failed too, the list of languages, profiles and styles is searched in order to find a member with the given name.

   The `'@'` at sign itself does not appear in the generated documentation. To display a single `'@'` at sign use a double `'@@'` at sign.

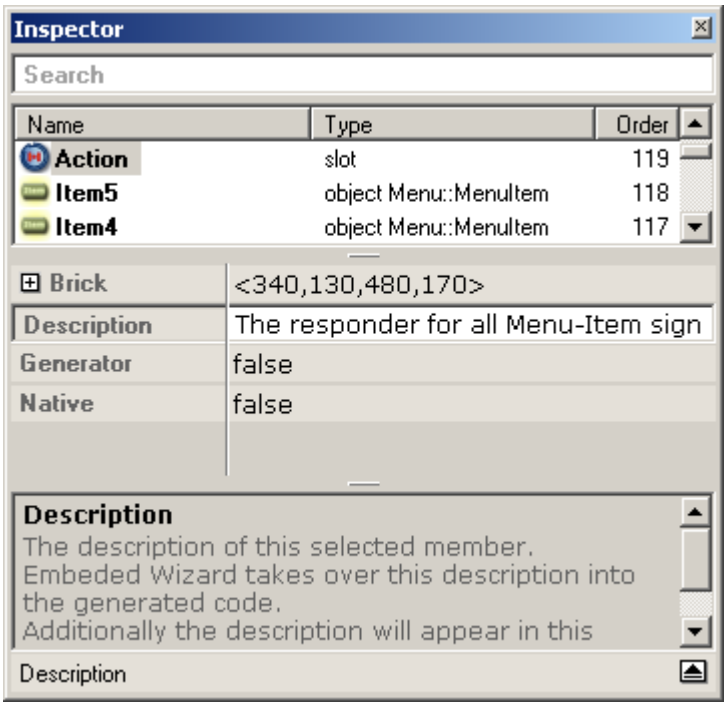
5. The `'-'` minus sign at the beginning of a text line is automatically converted in an HTML list item. Per default the list items will appear preceded with a big dot within the documentation file. For example:

```
// The method DoSomething() does following:
// - create a new menu object.
// - initialize the menu object.
// - display the menu on the screen.
method void DoSomething()
{
  ...
}
```


Copyright TARA Systems GmbH
www.embedded-wizard.de

The style of the generated documentation file is affected by the HTML cascaded style sheet file `'_stylesheet.css'`. This file is stored together with other template files in the directory `'<EmbeddedWizard>\HtmlDoc'`. You can adapt this file to your own preferences, so the used fonts, colors, text indent, borders, etc. can be changed. But the structure of the generated documentation can't be modified; it is fixed implemented in the Embedded Wizard application.

In the Embedded Wizard IDE the description of a member can be specified in its attribute Description within the Inspector window after the appropriate member has been selected:



# 13 Optimization

In order to generate compact and fast code for the target system, an optional optimization is available and can be explicitly enabled. The optimization level is controlled by the profile's attribute Optimization. Each profile within the project may define a different optimization strategy, depending on its Optimization attribute. The Optimization attribute may take one of the following values, also called optimization levels:

| Optimization level | Description |
|---|---|
| `None` | No optimization is performed. This is the default optimization level, which ensures the best possible compatibility to older versions of Embedded Wizard. |
| `Low` | Performs simple optimization steps:<br>➢ All invocations of non-overridden methods are performed directly, without the VMT (virtual method table) indirection.<br>➢ The read/write access to properties is performed directly, without the invocation of the onget or onset methods, if these methods contain only a single `pure` expression to access the property's value and these methods are not overridden in derived classes. |
| `Medium` | This optimization level forces the reorganization of the layout of Chora objects in order to reduce their size and the RAM usage.<br>The exact behavior of this optimization level depends on the Code Generator of the selected Platform Package. For example, in case of the 'ANSI C' Code Generator, the order of the object's members is modified, in order to compact small 8/16 bit variables together. |
| `High` | The highest level of optimization forces an automatic elimination of unused variables, methods, classes, constants, resources, etc. This optimization level results in the smallest and fastest GUI applications. This is because large parts of the Mosaic class library are designed as generic as possible. If not used, these parts are simply eliminated. In this same way unused parts of your GUI application can be eliminated. |

**Tabelle 13-1**

Depending on the set optimization level, the Embedded Wizards analyses the dependencies between the members of the project. It evaluates the logic of methods and determines, which members are unused or can be accessed in a more effective way. Then in the second pass the unused members are eliminated and the logic of the methods is adapted.


Copyright TARA Systems GmbH

All these modifications are done during the code generation only and they do not affect the content of your project; neither the eliminated variables are removed from your project nor the modifications done on the method's logic are taken over. These optimizations are temporary while the code is generated only.

The optimization works without trouble as long as the eliminated variables or methods are not accessed from the outside of the Chora application. For example, it is a usual practice to call Chora methods directly from the 'C' code in order to pass data from the target system to the GUI application. Due to the fact, that the Chora compiler does not know anything about the interface between the GUI application and the target system, it is not able to recognize which methods and variables are accessed from the outside of the Chora world. If not used elsewhere in the project, the affected members are simply eliminated. This results in unexpected and confusing 'C' compiler and linker errors.

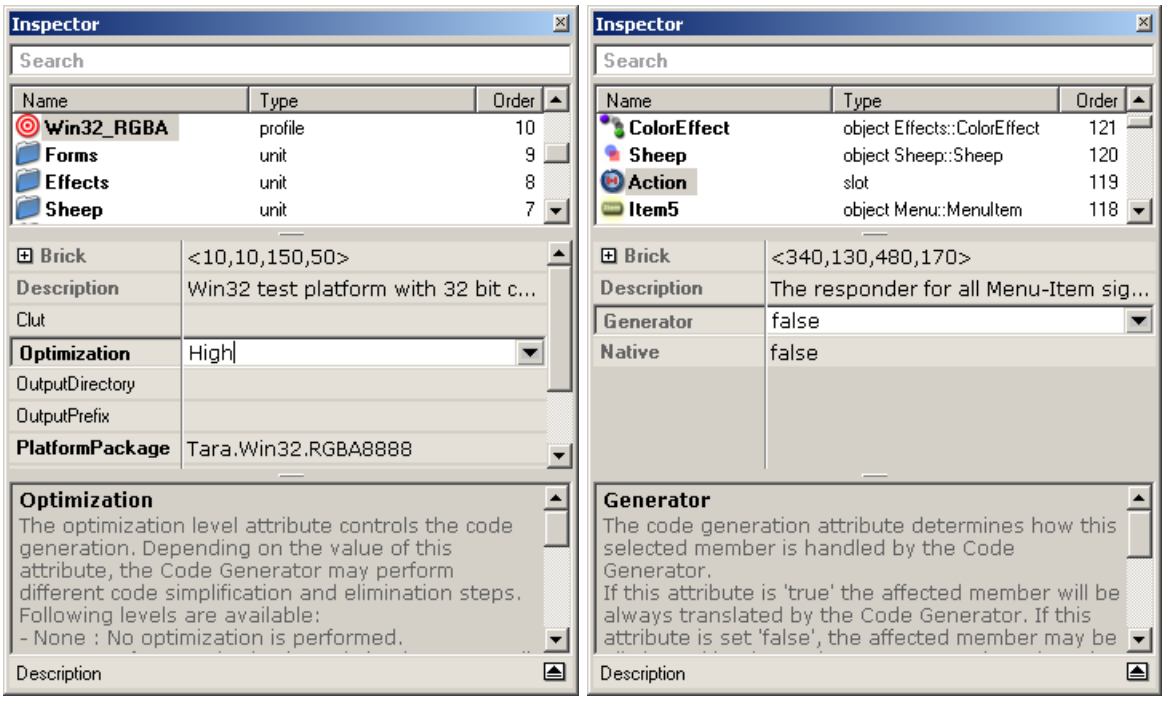
For this purpose, Embedded Wizard supports an optional `$output` directive. This directive can be placed in front of the affected class, method, variable, etc. definition in order to control the code generation for this member. The `$output true` directive forces, for example, the code generation independent of whether the affected member is used in the project or not. In contrast to this, the `$output false` directive disables the code generation unless the affected member is explicitly referenced from other (also used) parts of the application.

Therefore, it is necessary to sign all classes, variables, methods, etc. which are accessed from the outside of the GUI application with the directive `$output true`. All other classes, resources, constants, variables, methods, properties, etc. can be signed with `$output false` in order to reach the best possible elimination results. Usually it is enough, that the application class itself is forced to generate the code. All other members are invited automatically due to the existing dependencies from the application class. The application class is usually derived from the Mosaic class `Core::Root`.

For more details about the Optimization attribute and the `$output` directive see "Profiles and macros" (chapter 3.2) and "Control directives" (chapter 8.3).

In the Embedded Wizard IDE the Optimization attribute can be set in the Inspector window after the appropriate profile brick has been selected. The `$output` directive is available for all unit and class members in the Inspector window as the Generator attribute:


Copyright TARA Systems GmbH
www.embedded-wizard.de


Inspector
Search
Name
Type
Order
Win32_RGBA
profile
10
Forms
unit
9
Effects
unit
8
Sheep
unit
7
Brick
<10,10,150,50>
Description
Win32 test platform with 32 bit c...
Clut
Optimization
High
OutputDirectory
OutputPrefix
PlatformPackage
Tara.Win32.RGBA8888
Optimization
The optimization level attribute controls the code generation. Depending on the value of this attribute, the Code Generator may perform different code simplification and elimination steps. Following levels are available:
- None : No optimization is performed.
Description
Inspector
Search
Name
Type
Order
ColorEffect
object Effects::ColorEffect
121
Sheep
object Sheep::Sheep
120
Action
slot
119
Item5
object Menu::MenuItem
118
Brick
<340,130,480,170>
Description
The responder for all Menu-Item sig...
Generator
false
Native
false
Generator
The code generation attribute determines how this selected member is handled by the Code Generator.
If this attribute is 'true' the affected member will be always translated by the Code Generator. If this attribute is set 'false', the affected member may be
Description